

4ª EDIÇÃO
Revisada e ampliada



Google ANDROID

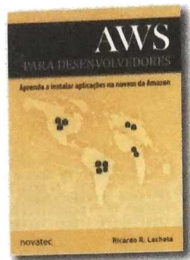
Aprenda a criar aplicações para dispositivos móveis
com o **Android SDK**

novatec

Ricardo R. Lecheta



Conheça também do mesmo autor



A Amazon Web Services (AWS) possui uma plataforma de computação em nuvem, repleta de serviços para auxiliar a criação de aplicações com alta disponibilidade e escalabilidade. O objetivo deste livro é apresentar os principais serviços da AWS, desde o básico ao avançado, em uma metodologia passo a passo e de forma prática. Você vai aprender a criar servidores virtuais na nuvem para hospedar seu próprio site, utilizar balanceadores de carga, escalonamento automático, monitoramento de serviços, bancos de dados na nuvem, armazenamento de arquivos, entrega de conteúdo estático e dinâmico, controle de permissões e segurança, serviço de e-mails, serviço de fila para mensagens assíncronas, mobile push, controle de custos etc.

ANDROID

Aprenda a criar aplicações para dispositivos móveis
com o **Android SDK**

4ª Edição

Ricardo R. Lecheta

Novatec

Copyright © 2009, 2010, 2013, 2015 da Novatec Editora Ltda.

Todos os direitos reservados e protegidos pela Lei 9610 de 19/02/1998.

É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Assistente editorial: Priscila A. Yoshimatsu

Revisão gramatical: Viviane Oshima

Editoração eletrônica: Carolina Kuwabata

Capa: Victor Bittow

ISBN: 978-85-7522-440-3

Histórico de impressões:

Agosto/2015	Primeira reimpressão
Junho/2015	Quarta edição (ISBN: 978-85-7522-440-3)
Outubro/2014	Quarta reimpressão
Abril/2014	Terceira reimpressão
Janeiro/2014	Segunda reimpressão
Setembro/2013	Primeira reimpressão
Março/2013	Terceira edição (ISBN: 978-85-7522-344-4)
Junho/2010	Segunda edição (ISBN: 978-85-7522-244-7)
Março/2009	Primeira edição (ISBN: 85-7522-186-0)

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

Email: novatec@novatec.com.br

Site: novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

OG20150729

Este livro é dedicado a toda a minha família e principalmente aos meus pais, Pedro e Maria Luisa, por terem me dado todo o carinho e a melhor educação possível, e por serem um grande exemplo de boas pessoas.

Em especial, este livro é dedicado à Paty, que é uma pessoa muito especial em minha vida e que sempre esteve ao meu lado me apoiando em todos os momentos. Paty, você sabe que só fico completo quando estamos juntos e você é a dona do meu coração. Te amo.

Sumário

Agradecimentos.....	21
Sobre o autor	22
Prefácio	23
Capítulo 1 ■ Introdução ao Android	25
1.1 Introdução.....	25
1.2 Open Handset Alliance e o Android.....	26
1.3 Sistema operacional Linux.....	27
1.4 Código aberto e livre	28
1.5 Máquina virtual Dalvik	28
1.6 Máquina virtual ART (Android Runtime).....	29
1.7 Conhecendo um pouco mais sobre o Android	29
1.8 Android Developer Challenge	30
1.9 Google Play.....	31
1.10 T-Mobile G1	31
1.11 Google Nexus.....	32
1.12 Um pouco sobre a história e versões do Android	32
1.13 Android 1.5 (Cupcake).....	33
1.14 Android 1.6 (Donut)	33
1.15 Android 2.0 e 2.1 (Eclair)	34
1.16 Android 2.2 (Froyo)	35
1.17 Android 2.3 (Gingerbread)	35
1.18 Android 3.0 (Honeycomb)	36
1.19 Android 4.0 (Ice Cream Sandwich)	36
1.20 Android 4.1 (Jelly Bean).....	37
1.21 Android 4.4 (KitKat).....	38
1.22 Android 5.0 (Lollipop)	38
1.23 Google I/O 2015 e o anúncio do Android M	40
Capítulo 2 ■ Configuração do ambiente de desenvolvimento.....	42
2.1 Android SDK.....	42
2.2 Requisitos de software e sistema	42

2.3 Plataforma (versão do Android)	43
2.4 Android Studio	44
2.5 Instalando os pacotes pelo SDK Manager	47
2.6 Intel Hardware Accelerated Execution Manager (HAXM)	50
2.7 Criando um projeto no Android Studio	51
2.8 Criando um emulador (AVD)	56
2.9 Executando o projeto no emulador	60
2.10 Algumas janelas importantes do Android Studio	61
2.11 Aplicações na tela principal (Home)	63
2.12 Entendendo um pouco mais sobre o emulador	64
2.13 ADB (Android Debug Bridge)	66
2.14 Informações básicas sobre a resolução do emulador	68
2.15 Como fazer o download dos exemplos do livro	70

Capítulo 3 • Conceitos básicos do Android..... 72

3.1 Estrutura do projeto no Android Studio	72
3.2 Arquivo AndroidManifest.xml	76
3.3 Classe MainActivity	78
3.4 Arquivo de layout activity_main.xml	80
3.5 Arquivo strings.xml	83
3.6 Classe R	83
3.7 Informações sobre como acessar recursos de texto e imagem	84
3.8 Arquivo build.gradle	86
3.9 LogCat – Escrevendo mensagens de log	88
3.10 Tratamento de eventos	90

Capítulo 4 • Activity..... 96

4.1 Activity	96
4.2 Classes FragmentActivity e AppCompatActivity	97
4.3 Ciclo de vida de uma activity	99
4.4 Ciclo de vida avançado – o que acontece ao rotacionar o celular?	105
4.5 Navegação entre telas e inicialização de uma nova activity	106
4.6 Mais detalhes sobre a classe Bundle e como passar parâmetros	114
4.7 O básico sobre action bar e como voltar para tela anterior	115
4.8 Links úteis	117

Capítulo 5 • Action Bar e temas..... 118

5.1 Introdução à Action Bar	118
5.2 Temas Holo e Material	118
5.3 Projeto de exemplo sobre action bar	119
5.4 Opções de visualização dos action buttons (always, never, ifRoom)	124
5.5 Template de ícones para os botões da action bar	127
	128

5.6 Classe android.app.ActionBar	129
5.7 SearchView	131
5.8 Action provider	133
5.9 Split action bar	135
5.10 Up navigation	137
5.11 Navegação por tabs na action bar	138
5.12 ActionBarCompat – a biblioteca de compatibilidade da action bar	141
5.13 Links úteis	146

Capítulo 6 ■ Interface gráfica – gerenciadores de layout 147

6.1 View	147
6.2 Classe ViewGroup	147
6.3 Configurando a altura e largura de uma view	148
6.4 Entendendo as constantes wrap_content e match_parent	149
6.5 FrameLayout	155
6.6 LinearLayout	157
6.7 LinearLayout – controle do alinhamento “layout_gravity”	158
6.8 LinearLayout – controle do peso	159
6.9 TableLayout – uso de uma tabela com linhas e colunas	163
6.10 TableLayout e shrinkColumns – contração de colunas	164
6.11 TableLayout e stretchColumns – expansão de colunas	165
6.12 TableLayout – criando um formulário	167
6.13 GridLayout	168
6.14 RelativeLayout	170
6.15 AbsoluteLayout (deprecated)	173
6.16 Utilizando layouts aninhados para criar telas complexas	173
6.17 Criação de um layout pela API – LinearLayout	174
6.18 Criação de um layout pela API – TableLayout	176
6.19 ScrollView	178
6.20 Alguns detalhes sobre a ActionBar e o “up navigation”	179
6.21 LayoutInflater – inflando arquivos XML	180
6.22 Links úteis	181

Capítulo 7 ■ Interface gráfica – View 182

7.1 Arquivo /res/values/strings.xml	182
7.2 Arquivo XML com as cores	183
7.3 Arquivo XML para criar um estilo CSS	184
7.4 Exemplo completo com estilos	185
7.5 View – A classe responsável por desenhar elementos na tela	187
7.6 TextView e EditText – campo de texto para digitar informações	188
7.7 AutoCompleteTextView	189
7.8 Button e ImageButton	191

7.9 CheckBox e ToggleButton	193
7.10 RadioButton	196
7.11 Spinner	200
7.12 ProgressDialog – janela de progresso	202
7.13 ProgressBar – barra de progresso	205
7.14 Toast – alertas rápidos	208
7.15 AlertDialog – alertas para o usuário confirmar	209
7.16 LayoutInflater – inflando um arquivo XML	210
7.17 ListView	210
7.18 ListView com adapter customizado	214
7.19 GridView	218
7.20 Gallery	221
7.21 ViewPager	223
7.22 ViewPager + TitleStrip ou TabStrip	228
7.23 ImageSwitcher	230
7.24 WebView	232
7.25 Movimentando uma imagem pela tela com touch	235
7.26 Desenho manual com a classe Canvas	238
7.27 Nunca utilize pixels	240

Capítulo 8 ■ Fragments 242

8.1 Como surgiram os fragments no Android 3.0 Honeycomb	242
8.2 Fragments é muito mais do que dividir a tela em duas partes	244
8.3 API de Fragments	249
8.4 Hello World fragment	250
8.5 Utilizando fragments com action bar + tabs	255
8.6 Utilizando fragments com action bar + tabs + ViewPager	258
8.7 Ciclo de vida de um fragment	261
8.8 Migrando um projeto que utiliza activity para fragments	264
8.9 Criando um layout dividido em partes nos tablets	272
8.10 Exemplos da API dos fragments	275
8.11 Back stack	278
8.12 Adicionando botões na action bar pelo fragment	279
8.13 Salvando o estado de um fragment	281
8.14 Vantagens de utilizar os fragments	283
8.15 Links úteis	283

Capítulo 9 ■ Animações 284

9.1 Drawable Animation	284
9.2 Classe Animation	285
9.3 View Animation	287
9.4 AlphaAnimation	287

9.5 RotateAnimation	289
9.6 ScaleAnimation	291
9.7 TranslateAnimation	293
9.8 AnimationSet	296
9.9 AnimationListener	298
9.10 Interpolator	298
9.11 O problema com a API de animações no Android 2.x	299
9.12 Property Animations	300
9.13 Classe ValueAnimator	301
9.14 Classe ObjectAnimator	303
9.15 ObjectAnimator – animação fade_in/fade_out	305
9.16 ObjectAnimator – animação de movimento	306
9.17 ObjectAnimator – animação de rotação	307
9.18 ObjectAnimator – animação de escala	308
9.19 AnimatorSet – criando um conjunto de animações	308
9.20 AnimatorListener	309
9.21 ViewPropertyAnimator – animação do jeito fácil	310
9.22 Classe ValueAnimator – outro exemplo	311
9.23 Aplicando animações no layout	311
9.24 Aplicando animações nos fragments	312
9.25 Aplicando animações ao navegar entre activities	313
9.26 NineOldAndroids – animações com compatibilidade	318
9.27 Links úteis	319

Capítulo 10 ■ Threads, Handler e AsyncTask 321

10.1 Introdução	321
10.2 Método sendMessage(msg)	324
10.3 Método post(runnable)	327
10.4 Atualizando a view dentro de uma thread	328
10.5 Agendando tarefas contínuas na activity	333
10.6 Implementação de um tela Splash Screen para sua aplicação	335
10.7 AsyncTask	337
10.8 Download de imagens com a biblioteca Picasso	342
10.9 Links úteis	344

Capítulo 11 ■ Material Design 345

11.1 Introdução	345
11.2 Tema Material	346
11.3 Paleta de cores	347
11.4 Elevação de views	349
11.5 Ripple – feedback ao toque	352
11.6 Floating Action Button (FAB)	355

11.7 CardView	357
11.8 RecyclerView.....	360
11.9 Efeito de revelação (Reveal Effect)	369
11.10 Extrair as cores de uma figura.....	371
11.11 Animações com item compartilhado entre duas activities	372
11.12 Compatibilidade com versões anteriores	378
11.13 Links úteis.....	378
Capítulo 12 ■ Toolbar	380
12.1 Introdução à Toolbar	380
12.2 Utilizando a Toolbar como a action bar	382
12.3 Utilizando a API da Toolbar (modo standalone)	386
12.4 Links úteis.....	388
Capítulo 13 ■ Navigation Drawer.....	389
13.1 Criando o projeto.....	389
13.2 Customizando as cores do tema Material	391
13.3 Criando a activity e o fragment base para o projeto	392
13.4 Classe Application – armazenando informações globais.....	394
13.5 Biblioteca android-utils.....	396
13.6 Como o Gradle encontrou a biblioteca android-utils	397
13.7 Configurando a Toolbar	398
13.8 Navigation Drawer.....	400
13.9 Criando o menu overflow	410
13.10 Navigation Drawer com Material Design	411
13.11 Material Design no Navigation Drawer	415
13.12 Criando os fragments do projeto	418
13.13 Links úteis.....	421
Capítulo 14 ■ WebView	422
14.1 Fragment com WebView	422
14.2 Swipe to Refresh	424
14.3 Interceptando requisições no WebView	426
14.4 Mostrando alertas com o AlertDialog	427
14.5 Executando JavaScript	431
14.6 Comunicação do JavaScript com a classe Android	431
14.7 Mostrando código HTML no WebView	433
14.8 Links úteis.....	433

Capítulo 15 ■ RecyclerView e tabs	434
15.1 Criando as classes de domínio	434
15.2 Criando a lista de carros	436
15.3 Tabs e ViewPager	442
15.4 Navegação de telas	447
15.5 Links úteis.....	451
 Capítulo 16 ■ Parser de XML, JSON e testes unitários	452
16.1 Lendo um arquivo local da pasta /res/raw	452
16.2 Parser de XML.....	453
16.3 Parser de JSON.....	457
16.4 Testes unitários no Android.....	460
16.5 Mais informações	462
 Capítulo 17 ■ Web services.....	463
17.1 Introdução	463
17.2 Requisição HTTP para consultar o web service	465
17.3 Utilizando a classe AsyncTask.....	466
17.4 Biblioteca simples para encapsular a AsyncTask.....	468
17.5 Atualização por Pull to Refresh	473
17.6 Verificando se existe conexão disponível.....	476
17.7 Requisições HTTP com Get e Post	479
17.8 Web services com WSDL	484
17.9 Links úteis	488
 Capítulo 18 ■ Persistência	490
18.1 Salvando as preferências do usuário com a classe SharedPreferences	490
18.2 Activity de configurações.....	493
18.3 Lendo e salvando arquivos	498
18.4 Trabalhando com arquivos na memória interna.....	498
18.5 Trabalhando com arquivos na memória externa (SD card).....	501
18.6 Outros métodos da classe Context.....	503
18.7 Brincando de fazer cache.....	504
18.8 Banco de dados SQLite	506
18.9 Criação de um banco de dados diretamente com a API	507
18.10 Inserção de registros no banco de dados	512
18.11 Atualização de registros no banco de dados.....	513
18.12 Exclusão de registros do banco de dados	514
18.13 Busca de registros no banco de dados.....	514
18.14 Métodos da classe Cursor	515
18.15 Continuando o projeto dos carros.....	517

18.16 Visualizando o banco de dados com a ferramenta adb	519
18.17 Visualizando o banco de dados com um cliente SQLite	520
18.18 Banco de dados versus web service	521
18.19 Adicionando ações na action bar	523
18.20 Editando o nome do carro	525
18.21 Excluindo um carro do banco de dados	530
18.22 Atualizando a lista com dados do web service	533
18.23 Modo de execução da activity “launchMode”	535
18.24 Fazendo backup na nuvem	536
18.25 Fazendo backup de um arquivo	540
18.26 Links úteis	541

Capítulo 19 = Action bar de contexto e compartilhamento 542

19.1 Introdução	542
19.2 Detectando toques longos – OnLongClickListener	543
19.3 Ativando o ActionMode na action bar	544
19.4 Removendo os carros selecionados	551
19.5 Compartilhando os carros selecionados	552
19.6 Compartilhando as fotos dos carros selecionados	555
19.7 Links úteis	559

Capítulo 20 = Intents 560

20.1 Intent – envio de uma mensagem ao Android	560
20.2 Intents explícitas e implícitas	561
20.3 Exemplos de intents nativas	562
20.4 Permissões	568
20.5 Retornando resultados de uma intent – startActivityForResult	568
20.6 IntentFilter	570
20.7 Por que a MainActivity declara um <intent-filter>?	572
20.8 Exemplo completo com intent customizada	572
20.9 Verificando se uma intent será encontrada	580
20.10 Interceptando aplicações nativas	581
20.11 Lendo código de barras	583
20.12 Nomenclatura das intents	585
20.13 Links úteis	585

Capítulo 21 = Multimídia – áudio, vídeo e câmera 586

21.1 Formatos de áudio e vídeo suportados	586
21.2 Classe Media Player	586
21.3 Criando um player de mp3	589
21.4 Reproduzindo vídeo com a classe VideoView	593
21.5 Utilizando uma intent e o player de vídeo nativo	595

21.6 Utilizando o VideoView no projeto dos carros	600
21.7 Tirando fotos com uma intent	604
21.8 Tirando fotos – como obter o arquivo da foto.....	607
21.9 Trabalhando com bitmaps de forma eficiente	610
21.10 Enviando a imagem para o servidor.....	612
21.11 Gravando áudio e vídeo	612
21.12 Links úteis	614

Capítulo 22 ■ Mapas 615

22.1 Introdução.....	615
22.2 Google Maps Android API – Versão 2.....	616
22.3 Google Play Services	616
22.4 Gerando a chave de acesso dos mapas	617
22.5 Configurando o projeto	620
22.6 Adicionando o mapa no projeto dos carros.....	624
22.7 Classe GoogleMap	628
22.8 Localização do mapa – latitude e longitude	629
22.9 CameraPosition – zoom.....	630
22.10 Configurando o tipo do mapa.....	631
22.11 Colocando os conceitos em práticas	632
22.12 CameraPosition.....	635
22.13 CameraPosition – bearing “rotação”	636
22.14 CameraPosition – tilt “inclinação”	637
22.15 Monitorando os eventos do mapa	638
22.16 Marcadores.....	639
22.17 Polyline – desenhando uma linha no mapa	644
22.18 Links úteis	645

Capítulo 23 ■ Google Play Services e localização 646

23.1 Introdução.....	646
23.2 My Location.....	647
23.3 Monitorando o GPS (à moda antiga)	647
23.4 Monitorando o GPS (Fused Location Provider)	649
23.5 Conectando-se ao Google Play Services	650
23.6 Obtendo a última localização de forma eficiente	651
23.7 API de localização do Google Play Services	655
23.8 Desenhando uma rota entre dois pontos.....	659
23.9 Buscando um endereço	661
23.10 Links úteis	661

Capítulo 24 ■ BroadcastReceiver	662
24.1 Introdução.....	663
24.2 Configurando um receiver de forma estática	665
24.3 Configurando um receiver de forma dinâmica	667
24.4 Quando utilizar um receiver estático ou dinâmico?	668
24.5 Classe LocalBroadcastManager	668
24.6 Execução de um receiver ao inicializar o sistema operacional.....	669
24.7 Interceptando uma mensagem SMS.....	670
24.8 Ciclo de vida	671
24.9 Delegando o trabalho para um service.....	671
24.10 Mostrando uma notificação para o usuário	672
24.11 Links úteis.....	
Capítulo 25 ■ Notification	673
25.1 Por que usar uma notificação para se comunicar com o usuário	673
25.2 Criando uma notificação simples	674
25.3 Heads-up notifications.....	680
25.4 Notificações na tela de bloqueio	682
25.5 Criando uma notificação grande (big view notifications).....	684
25.6 Criando uma notificação com ações	686
25.7 Cancelando uma notificação	688
25.8 Mais informações sobre a classe PendingIntent	688
25.9 Exemplo com notificação e BroadcastReceiver.....	689
25.10 Mostrando uma barra de progresso na notificação.....	692
25.11 Links úteis.....	693
Capítulo 26 ■ AlarmManager	694
26.1 Por que utilizar um alarme (agendar uma tarefa)	694
26.2 Método da classe AlarmManager	695
26.3 Agendando um alarme	697
26.4 Repetindo o alarme.....	701
26.5 Classe Calendar	701
26.6 Quando utilizar ou não um alarme	702
26.7 Links úteis	703
Capítulo 27 ■ Service e JobInfo.....	704
27.1 Introdução	704
27.2 Exemplos de serviços.....	705
27.3 Como iniciar e parar um serviço.....	706
27.4 Exemplo prático	707
27.5 Deixar o serviço executando depois de sair de uma tela.....	712

27.6 Entendendo o ciclo de vida de um serviço	712
27.7 A classe IntentService	714
27.8 Criando um player mp3	716
27.9 Método bindService(intent,con,flags)	723
27.10 Qual método utilizar para iniciar um serviço?.....	725
27.11 Um serviço em execução contínua não consome muito processamento? ..	725
27.12 JobInfo – a nova API do Lollipop	726
27.13 Links úteis	730

Capítulo 28 ■ GCM – Google Cloud Messaging..... 731

28.1 O que é push?	731
28.2 Como funciona o GCM.....	732
28.3 Gerando a chave de acesso do GCM.....	733
28.4 Obtendo o Project Number.....	735
28.5 Executando o projeto de exemplo	735
28.6 Enviando a mensagem de push	736
28.7 Criando o projeto Android passo a passo	739
28.8 Classe GoogleCloudMessaging.....	740
28.9 Configurando o projeto Android.....	742
28.10 Criando a activity para fazer o registro no GCM.....	746
28.11 Links úteis	750

Capítulo 29 ■ Salvando o estado da aplicação 751

29.1 Troca de configurações – configuration changes.....	751
29.2 Salvando o estado com o método onSaveInstanceState(bundle)	752
29.3 Salvando o estado com o método setRetainInstance(boolean)	755
29.4 A importância de reter a instância do fragment	756
29.5 Manter uma thread executando durante a troca de orientação	757
29.6 Como bloquear a troca de orientação	761
29.7 Dispositivos com teclado.....	762
29.8 Configuração android:configChanges e o método onConfigurationChanged ...	762
29.9 Salvando o estado no projeto dos carros	765
29.10 Links úteis	765

Capítulo 30 ■ Suportando diferentes tamanhos de telas 766

30.1 Unidades de medida	766
30.2 Tamanho de tela (screen size)	767
30.3 Proporção da tela (aspect ratio)	768
30.4 Resolução e densidade da tela	769
30.5 O problema de utilizar pixels	769
30.6 DIP ou DP (density-independent pixel)	771
30.7 Tabela de densidade dos dispositivos	773

30.8 Customizando as imagens conforme a densidade da tela	774
30.9 Trabalhando com a unidade dp no Canvas	775
30.10 Tamanho da tela em dp	777
30.11 Dimensões (dimen)	780
30.12 Qualificadores de recursos para tablets	781
30.13 Links úteis	
Capítulo 31 ■ Threads avançado – AsyncTask e Loader	783
31.1 O problema com o ProgressDialog	783
31.2 Controlando a troca de orientação ao executar uma task	788
31.3 Executando a AsyncTask de forma serial ou paralela	793
31.4 Loader	794
31.5 Opinião do autor	808
31.6 Links úteis	809
Capítulo 32 ■ Agenda de contatos e content provider	810
32.1 Por que utilizar a classe ContentProvider “provedor de conteúdo”	810
32.2 URI – Immutable URI reference	811
32.3 Exemplos de provedores de conteúdo nativos	812
32.4 Lendo os contatos da agenda	814
32.5 Como ler todos os telefones e a foto de um contato	816
32.6 Mostrando os contatos em um ListView	821
32.7 Utilizando um CursorAdapter	824
32.8 Utilizando um CursorLoader	827
32.9 Monitorando a fonte de dados com um loader	828
32.10 Criando um provedor de conteúdo customizado	830
32.11 Classe ContentProvider	831
32.12 Classe estática Carros	838
32.13 Links úteis	840
Capítulo 33 ■ SMS	841
33.1 Enviando SMS por intent ou pela API	841
33.2 Criando um BroadcastReceiver para receber um SMS	845
33.3 Links úteis	848
Capítulo 34 ■ Gestos	849
34.1 Introdução	849
34.2 Reconhecendo gestos previamente cadastrados	849
34.3 Detectando gestos comuns, como scroll lateral	857
34.4 Detectando gesto de pinch (zoom)	861
34.5 Links úteis	866

Capítulo 35 ■ Sensores e Google Fit	867
35.1 Como obter a lista de sensores disponíveis	867
35.2 Testando os sensores	871
35.3 Sensor de luminosidade	876
35.4 Sensor de temperatura	878
35.5 Sensor de proximidade	878
35.6 Sensor de acelerômetro	879
35.7 Movendo uma view pela tela com o acelerômetro	886
35.8 Google Fit	889
35.9 Links úteis.....	898
 Capítulo 36 ■ Bluetooth	899
36.1 Verificando se o dispositivo suporta Bluetooth	899
36.2 Ativando o Bluetooth por programação	900
36.3 Listando os dispositivos pareados	902
36.4 Buscar novos dispositivos Bluetooth	902
36.5 Deixando o Bluetooth visível para ser encontrado.....	906
36.6 Criando um BluetoothDevice pelo endereço.....	907
36.7 Chat em Bluetooth	908
36.8 Conectando-se ao Bluetooth pela serial	918
36.9 Links úteis	919
 Capítulo 37 ■ Reconhecimento de voz	920
37.1 Introdução	920
37.2 Hello TTS – faça seu Android falar	921
37.3 Verificando o idioma e falando em português.....	927
37.4 Reconhecimento de voz por intent	929
37.5 Reconhecimento de voz por um listener	932
37.6 Links úteis.....	935
 Capítulo 38 ■ Gradle	936
38.1 Introdução.....	936
38.2 Gerenciando dependências	937
38.3 Trabalhando com módulos	938
38.4 Trabalhando com bibliotecas	941
38.5 Criando uma biblioteca	941
38.6 Configurando um servidor Maven Sonatype Nexus	946
38.7 Publicando no Maven Central.....	946
38.8 Flavors	947
38.9 Classe BuildConfig.....	951

38.10 Assinando o aplicativo para o build release	953
38.11 Links úteis	955

Capítulo 39 ■ Android Wear..... 957

39.1 Introdução	957
39.2 Hello World Wear	959
39.3 Conectando o smartphone no Android Wear	963
39.4 Conectando o smartphone no relógio físico	965
39.5 Notificações no wear	966
39.6 Notificações com várias páginas	967
39.7 Notificações empilhadas	968
39.8 Notificações com comandos de voz.....	970
39.9 Google Play Services e Wearable API.....	972
39.10 Node API	973
39.11 Message API	974
39.12 Data API	975
39.13 Enviando mensagens entre o smartphone e o Wear	976
39.14 Enviando uma foto tirada pela câmera para o wear	986
39.15 Criando views e layouts para wear.....	989
39.16 Criando cards (cartões).....	990
39.17 Criando listas.....	995
39.18 Criando páginas (ViewPager)	997
39.19 Criando páginas em grid (GridViewPager)	999
39.20 Aplicativos em tela cheia (Full-Screen).....	1003
39.21 Animação de confirmação.....	1005
39.22 Alertas de sucesso e erro	1007
39.23 Interceptando eventos em background	1008
39.24 Localização e sensores	1009
39.25 Links úteis.....	1010

Capítulo 40 ■ Google Play 1011

40.1 Controle da versão de sua aplicação.....	1011
40.2 Compilando o projeto corretamente.....	1012
40.3 Assinando o aplicativo pelo Android Studio/Gradle.....	1013
40.4 Publicando no Google Play	1013
40.5 Monetização com anúncios.....	1014
40.6 Links úteis	1016

Agradecimentos

Este livro não teria acontecido sem a ajuda de diversas pessoas, algumas pela contribuição técnica e outras pela motivação.

Agradeço a todo o pessoal da Wasys, Tivit e Livetouch pelos incríveis projetos de mobilidade, e a todo o pessoal técnico da equipe pelas ideias e sugestões.

Agradeço a toda a comunidade Android pelo feedback e incentivo para continuar escrevendo e lançar esta nova edição.

Em especial, agradeço ao Rubens Prates, editor da Novatec, por toda a calma e orientação em todas as etapas da produção deste livro. Seus conselhos foram fundamentais para tudo isso acontecer.

Agradeço à Ana Carolina Prates, da Novatec, por todo o apoio de sempre.

Por último, agradeço a você pelo interesse em ler esta obra. Espero que a leitura seja simples e empolgante.

Sobre o autor

Ricardo R. Lecheta é formado em Ciência da Computação e pós-graduado em Gestão do Desenvolvimento de Software pela PUCPR. Tem certificações da Sun, IBM e Rational, entre elas SCMD (J2ME) e SCEA (Arquiteto).

Já desenvolveu projetos em JEE e .Net, além de mobile, para grandes empresas como Ambev, HSBC, Itaú, Rede, Renault, Nissan, Coca-Cola, Unimed, Boticário, Banco Bonsucesso, Bovespa, UOL, Globo, Mondial, Agência Estado, Cosan, Metalfrio, Polícia Federal, entre outras, e nos últimos anos vem se especializando na integração de sistemas legados com a plataforma mobile.

Atualmente trabalha com desenvolvimento e consultoria de tecnologias mobile para diversas plataformas e pode ser contatado pelo email rlcheta@gmail.com e no facebook.com/ricardolecheta.

Prefácio

Assim que a primeira versão do SDK (ambiente de desenvolvimento) do Android foi lançada, diversos sites da internet sobre tecnologia da informação já anunciaram que o Google estava lançando uma nova plataforma completa e totalmente aberta para dispositivos móveis. A notícia percorreu o mundo, e era só o que se comentava em todos os lugares. Todos discutiam se o Android ocuparia seu espaço no mercado e quais seriam suas vantagens sobre os concorrentes. O fato de ser lançado pelo Google, o gigante da internet, causou ainda mais curiosidade e expectativa de todos os lados, atraindo a atenção de muita gente, desde usuários comuns a grandes empresas e desenvolvedores em geral.

Anos depois do lançamento, podemos constatar que realmente o Android veio para ficar e vem constantemente revolucionando o mercado de mobilidade. Atualmente o Android está disponível para diversas plataformas, como smartphones e tablets, TV (Google TV), relógios (Android Wear), óculos (Google Glass), carros (Android Auto) etc., e é o sistema operacional móvel mais utilizado no mundo.

O objetivo deste livro é apresentar ao leitor este novo e fascinante mundo do Android, que está revolucionando o desenvolvimento de aplicações para celulares. Para ler esta obra, é recomendado um bom entendimento da linguagem Java e experiência com o desenvolvimento de aplicações em geral. Cada capítulo do livro tem um projeto de exemplo, que está disponível para download gratuitamente no site www.livroandroid.com.br.

Os capítulos 1, 2 e 3 deste livro são introdutórios sobre a arquitetura básica do Android e explicam como instalar o SDK e configurar o ambiente de desenvolvimento no Android Studio. Os capítulos 4 a 12 explicam recursos importantes disponíveis na plataforma, fornecendo uma base sólida sobre a estrutura de uma aplicação Android.

Do capítulo 13 em diante vamos estudar diversos conceitos de forma prática, desenvolvendo o aplicativo dos carros passo a passo. O objetivo é que você aprenda na prática por meio de exemplos, variando do básico ao avançado. O livro também está atualizado para a última versão do Android e padrões do Material Design.

No final ainda temos capítulos específicos sobre o novo sistema de builds do Android (Gradle), desenvolvimento de aplicativos para relógios (Android Wear) e como publicar um aplicativo no Google Play.

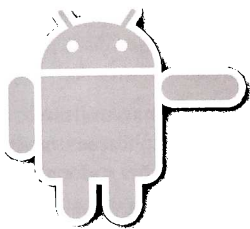
Espero que a leitura deste livro seja simples e ao mesmo tempo empolgante, que os exemplos e explicações possam lhe fornecer uma base sólida para desenvolver ótimas aplicações e que você possa aproveitar ao máximo o mercado de mobile, que não para de crescer e está sempre à procura de bons profissionais.

Ricardo Lecheta

<http://facebook.com/ricardolecheta>

<https://plus.google.com/+RicardoLecheta>

<https://twitter.com/rlecheta>



CAPÍTULO 1

Introdução ao Android

1.1 Introdução

Nos dias de hoje, ninguém consegue ficar longe de um celular, seja para mandar um email, tirar uma foto, assistir um vídeo, conversar com os amigos, navegar na internet, acompanhar as redes sociais etc. Portanto, os smartphones e tablets atualmente são objetos praticamente inseparáveis da maioria das pessoas.

Segundo pesquisas, mais de 3 bilhões de pessoas têm um telefone celular, e o mercado de aplicativos virou uma febre, rendendo bilhões todos os anos.

Nesse mercado competitivo, temos vários lados da moeda. Os usuários comuns buscam um celular com um visual elegante, moderno, de fácil navegação, assim como uma infinidade de aplicativos e recursos. Tanto as empresas quanto os desenvolvedores buscam uma plataforma moderna e ágil para desenvolver aplicativos. Os fabricantes (LG, Motorola, Samsung, HTC, Intel, Sony etc.) precisam de uma plataforma robusta e rica em funcionalidades para lançar no mercado os seus produtos. É aqui onde o Android se encaixa, pois ele é perfeito para todos os casos.

O Android é o sistema operacional móvel do Google e atualmente é líder mundial nesse segmento. No entanto, o sucesso do Android não se deve apenas à força do Google – por trás do desenvolvimento de toda a plataforma estão gigantes do mercado de mobilidade, como fabricantes de celulares e operadoras. Esse grupo que ajuda no desenvolvimento da plataforma é chamado de OHA (Open Handset Alliance) e conta com nomes de peso como Intel, Samsung, LG, Motorola, Sony Ericsson, HTC, Sprint Nextel, ASUS, Acer, Dell, Garmin etc. Existe todo um ecossistema interessado no desenvolvimento de uma plataforma móvel poderosa e flexível, de código-aberto e que atenda às necessidades de todos. Embora o Google represente grande parte da força do Android, com certeza a plataforma está hoje onde está devido à ajuda de outras potências do mercado móvel.

Atualmente o Android está disponível para diversas plataformas, como smartphones e tablets, TV (Google TV), relógios (Android Wear), óculos (Google Glass), carros (Android Auto) e é o sistema operacional móvel mais utilizado no mundo.

Vale lembrar que o mercado corporativo também está cada vez mais utilizando o mobile, tanto que diversas empresas estão buscando incorporar aplicações móveis a seu dia a dia para agilizar seus negócios e integrar as aplicações móveis com seus sistemas de back-end. Empresas obviamente visam o lucro; por isso, tanto os smartphones quanto os tablets ocupam um importante espaço em um mundo em que a palavra “mobilidade” está cada vez mais conhecida.

Dentro desse contexto, estamos diante de uma excelente oportunidade, pois o mobile é um grande pilar na área de tecnologia e segundo pesquisas é uma das áreas que mais vai crescer nos próximos anos, por isso você não pode ficar fora dessa.

O objetivo deste livro é explicar o desenvolvimento de aplicativos para Android, do básico ao avançado, com diversos exemplos práticos e dicas de que você vai precisar no dia a dia.

1.2 Open Handset Alliance e o Android

A Open Handset Alliance (OHA) é um grupo formado por gigantes do mercado de telefonia de celulares liderados pelo Google. Entre alguns integrantes do grupo estão nomes consagrados como Intel, HTC, LG, Motorola, Samsung, Sony Ericsson, Toshiba, HTC, Huawei, Sprint Nextel, China Mobile, T-Mobile, ASUS, Acer, Dell, Garmin e muito mais.

Quando este livro foi escrito, o grupo era formado por 84 integrantes de peso, e você pode verificar a lista completa e atualizada em: www.openhandsetalliance.com/oha_members.html.

No site da OHA existe uma ótima descrição do que é essa aliança. O texto está em inglês e vou apenas traduzir uma breve citação aqui. “Hoje, existem 1,5 bilhão de aparelhos de televisão em uso em todo o mundo e 1 bilhão de pessoas têm acesso à internet. No entanto, quase 3 bilhões de pessoas têm um telefone celular, tornando o aparelho um dos produtos de consumo mais bem-sucedidos do mundo. Dessa forma, construir um aparelho celular superior melhoraria a vida de inúmeras pessoas em todo o mundo. A Open Handset Alliance é um grupo formado por empresas líderes em tecnologia móvel que compartilham essa visão para mudar a experiência móvel de todos os consumidores”.

Assim, o objetivo do grupo é definir uma plataforma única e aberta para celulares para deixar os consumidores mais satisfeitos com o produto final. Outro

objetivo principal dessa aliança é criar uma plataforma moderna e flexível para o desenvolvimento de aplicações corporativas. O resultado dessa união, como você já deve saber, foi o nascimento do Android.

Todos acabam se beneficiando com os avanços alcançados pelo grupo OHA e a plataforma do Android: os fabricantes de celulares, os usuários comuns e, é claro, as empresas em geral e os desenvolvedores de aplicações.

Os usuários de celulares são extremamente favorecidos com tudo isso. Hoje em dia, todos querem um celular com um bom visual, de fácil usabilidade, com tela touch screen, câmera, músicas, jogos, GPS, acesso à internet e muito mais, e o celular cada vez mais ocupa um espaço importante na vida das pessoas. O Android foi criado justamente para agradar esses usuários, possibilitando que encontrem todos os recursos esperados em apenas um aparelho. O mundo da tecnologia está sempre em evolução, e a OHA tem como objetivo principal manter uma plataforma-padrão na qual todas as novas tendências do mercado estejam englobadas em uma única solução.

Para os fabricantes de celulares, o fato de existir uma plataforma única e consolidada é uma grande vantagem para criar novos aparelhos. A grande vantagem para eles é que a plataforma também é livre e de código aberto. A licença do Android é flexível e permite que cada fabricante possa realizar alterações no código-fonte para customizar seus produtos, e, o melhor de tudo, sem necessidade de compartilhar essas alterações com ninguém. O Android também é “free”, portanto, os fabricantes podem usar e abusar dele sem precisar pagar por isso.

O fato de o Android ser de código aberto contribui muito para seu aperfeiçoamento, uma vez que desenvolvedores de todos os lugares do mundo podem contribuir para seu código-fonte, adicionando novas funcionalidades ou simplesmente corrigindo falhas.

Já os desenvolvedores de aplicações podem desfrutar de uma plataforma de desenvolvimento moderna com diversos recursos incríveis, com tudo o que há de mais moderno. Este é o tema deste livro: o desenvolvimento de aplicações com o Android. E aqui você vai entender o porquê de toda essa revolução.

1.3 Sistema operacional Linux

O sistema operacional do Android é baseado no kernel do Linux, que é responsável por gerenciar a memória, os processos, threads, segurança dos arquivos e pastas, além de redes e drivers.

Cada aplicativo no Android dispara um novo processo no sistema operacional. Alguns deles podem exibir uma tela para o usuário, e outros podem ficar em execução em segundo plano por tempo indeterminado. Diversos processos e aplicativos podem ser executados simultaneamente, e o kernel do sistema operacional é o responsável por realizar todo o controle de memória.

Caso necessário, o próprio sistema operacional pode decidir encerrar algum processo para liberar memória e recursos, e talvez até reiniciar o mesmo processo posteriormente quando a situação estiver controlada.

Toda a segurança do Android é baseada na segurança do Linux. No Android, cada aplicação é executada em um único processo e cada processo por sua vez contém uma thread dedicada. Para cada aplicação instalada no celular é criado um usuário no sistema operacional para ter acesso a sua estrutura de diretórios. Dessa forma, nenhum outro usuário pode ter acesso a essa aplicação.

1.4 Código aberto e livre

O Android é a primeira plataforma para aplicações móveis completamente livre e de código aberto (open source), o que representa uma grande vantagem competitiva para sua evolução, uma vez que diversas empresas e desenvolvedores do mundo podem contribuir para melhorar a plataforma.

Para os fabricantes de celulares, isso também é uma grande vantagem, uma vez que é possível utilizar o sistema operacional do Android em seus celulares sem ter de pagar por isso. Além disso, a licença Apache Software Foundation (ASF) permite que alterações sejam efetuadas no código-fonte para criar produtos customizados sem precisar compartilhar as alterações com ninguém.

Você pode obter mais informações e até fazer o download do código-fonte do Android no seguinte site: <http://source.android.com/>.

1.5 Máquina virtual Dalvik

Provavelmente você já sabe que a linguagem Java é utilizada para construir as aplicações para o Android. O fato é que em seu sistema operacional não existe uma máquina virtual Java (JVM). Na verdade, o que temos é uma máquina virtual chamada Dalvik que é otimizada para execução em dispositivos móveis.

Ao desenvolver as aplicações para o Android, você vai utilizar a linguagem Java e todos os seus recursos normalmente, mas depois que o bytecode (.class) é

compilado ele é convertido para o formato *.dex* (Dalvik Executable), que representa a aplicação do Android compilada.

Depois disso, os arquivos *.dex* e outros recursos como imagens são compactados em um único arquivo com a extensão *.apk* (Android Package File), que representa a aplicação final, pronta para ser distribuída e instalada. Ao utilizar o ambiente de desenvolvimento do Android Studio, toda essa compilação e geração do arquivo *.apk* ocorre automaticamente, portanto, não é preciso se preocupar com isso.

Atualmente, o sistema de build utilizado é o Gradle, o qual é independente do Android Studio e pode ser executado separadamente. Portanto, você pode compilar todo o código por linha de comando se necessário.

1.6 Máquina virtual ART (Android Runtime)

A partir do Android 4.4 (KitKat) foi criada a máquina virtual ART (Android Runtime) com o objetivo de substituir a Dalvik, e naquela época o ART podia ser ativado opcionalmente nas configurações. Quando foi lançado o Android 5.0 (Lollipop), o ART se tornou a máquina virtual padrão, substituindo a Dalvik.

Uma das melhorias do ART é a compilação Ahead-of-time (AOT), que tem o objetivo de otimizar o código ao máximo para melhorar o desempenho do aplicativo. O ART também tem um melhor funcionamento do Garbage Collector (GC) e apresenta melhorias no suporte ao debug de aplicativos.

Na prática os desenvolvedores ou usuários não são afetados se o sistema está utilizando a Dalvik ou ART, mas o Google afirma que o ART apresenta um desempenho muito melhor.

1.7 Conhecendo um pouco mais sobre o Android

Todo celular tem uma tela inicial com alguns ícones e um menu, certo? Todo celular também tem uma agenda de contatos e uma tela para fazer a ligação, não é?

Agora, você já pensou em trocar algumas dessas telas por uma tela customizada desenvolvida por você? Com o Android isso é possível. Sua arquitetura é muito flexível e você pode integrar aplicações nativas com sua aplicação, ou até mesmo substituir qualquer aplicação nativa existente por uma que você mesmo criou. É isso que muitos fabricantes e operadores fazem ao customizar os aparelhos.

É possível integrar aplicações de uma forma simples, sejam elas desenvolvidas por você, sejam aplicações nativas. Por exemplo, imagine que sua aplicação

precise consultar a agenda de contatos para selecionar determinado amigo (ex.: WhatsApp), e logo depois visualizar o endereço dele em um mapa. Bem, que existe a agenda de contatos e o Google Maps no Android todos já sabem, mas será que é possível utilizá-los e integrá-los em nossas aplicações? A resposta é sim. Integração é uma das palavras-chaves em aplicações corporativas, e a arquitetura do Android foi criada justamente pensando nisso.

Nota: o Android tem muitos diferenciais interessantes e uma arquitetura realmente flexível focada na integração de aplicações. Não existe diferença entre uma aplicação nativa e uma desenvolvida por você.

Falando em integração, existe uma classe que é o coração do Android, chamada de Intent, a qual vamos estudar no livro. Essa classe nada mais é do que uma mensagem enviada ao sistema operacional informando nossa “intenção” de realizar determinada tarefa.

Então, no sistema operacional do Android, mensagens são disparadas para todos os lados, identificadas pela classe Intent. Conforme o conteúdo da mensagem, ela pode ser interceptada por qualquer aplicação interessada a fim de realizar a tarefa que for necessária. Por exemplo, se você deseja abrir uma aplicação nativa como o browser ou abrir uma nova tela de sua aplicação, a única coisa que você precisa fazer é criar esse objeto Intent e configurar o conteúdo de sua mensagem corretamente para ser interpretado pelo sistema operacional.

Outro ponto forte do Android é que seu sistema operacional é baseado no Linux, o qual se encarrega de gerenciar a memória e os processos. Isso permite que diversas aplicações possam ser executadas ao mesmo tempo, de forma que as aplicações em segundo plano consigam executar sem atrapalhar a atividade do usuário, enquanto ele está acessando a internet ou atendendo uma ligação.

É claro que não podemos nos esquecer dos recursos visuais e todas as APIs disponíveis, mas o fato de o Android ser aberto e totalmente customizado é um diferencial que vale a pena ressaltar.

1.8 Android Developer Challenge

Agora vamos falar um pouco da história do sistema operacional do robzinho verde. Para promover o Android, o Google começou investindo pesado e, assim que a primeira versão do SDK foi lançada, também foi anunciado o famoso concurso Android Developer Challenge (ADC), com mais de US\$ 10 milhões em prêmios.

Apenas por curiosidade, eu já trabalhava com mobile desde 2001 e foi nesse momento que me encantei com o Android e comecei a escrever a 1ª edição deste livro, que ficou pronta em 2009, pouco depois da 1ª fase deste concurso terminar.

O prazo para enviar as aplicações do ADC era 14 de abril de 2008, e o concurso foi dividido em duas fases. Na primeira fase, as 50 melhores aplicações recebiam US\$ 25 mil e, na segunda, mais 20 das melhores aplicações seriam selecionadas para receber US\$ 275 mil, e algumas US\$ 100 mil.

Na primeira etapa, as aplicações foram testadas no próprio emulador do Android, porque na época nenhum celular com o Android tinha sido lançado. Isso foi uma grande sacada do Google para melhorar a plataforma e ajudar a testá-la, sendo que desenvolvedores de todo o mundo estavam interessados em desenvolver as aplicações para talvez faturar uma bolada. Esse concurso literalmente agitou o mundo todo, com isso o Google conseguiu testar o SDK e consolidar seu produto.

A segunda parte do concurso foi anunciada para acontecer somente depois que o primeiro celular com o Android fosse lançado, dessa vez as aplicações seriam testadas em um aparelho real e não mais em um emulador.

1.9 Google Play

Para auxiliar a distribuição das aplicações do Android, além da divulgação de sua nova plataforma, foi criado o site Google Play (<https://play.google.com>), que inicialmente se chamava Android Market. O objetivo do site é fornecer aos desenvolvedores de aplicativos um lugar comum para disponibilizar suas aplicações.

Para publicar uma aplicação, o desenvolvedor precisa pagar a taxa de US\$ 25 (o pagamento é feito uma única vez por meio de um cartão de crédito internacional) e concordar com os termos de uso. Depois disso, o aplicativo já pode ser publicado e instalado pelos usuários. Existem aplicativos que são gratuitos, enquanto outros são pagos. Uma boa notícia para os desenvolvedores é que 70% dos lucros com os aplicativos vendidos serão repassados para quem os construiu. Para mais informações, visite o site do console do desenvolvedor no seguinte endereço.

<https://play.google.com/apps/publish/>

1.10 T-Mobile G1

O T-Mobile G1 desenvolvido pela HTC foi o primeiro celular lançado com a plataforma do Android e, como esperado, agitou o mercado. A notícia de seu

lançamento causou um grande impacto e superou as expectativas de vendas da HTC: mesmo antes de seu lançamento, todo o estoque para os pedidos de pré-venda já havia sido esgotado.

Os primeiros celulares HTC G1 começaram a ser vendidos nos Estados Unidos no dia 22 de outubro de 2008 por US\$ 179. Um fato interessante é que eu terminei a 1ª edição deste livro antes mesmo de o G1 ser lançado e fiz todos meus estudos somente utilizando o emulador. Lembro que no 1º curso de Android que ministrei alguns alunos tinham comprado o G1 e vieram me mostrar. Foi emocionante.

1.11 Google Nexus

Desde o HTC G1 até os dias de hoje, o Android não parou de evoluir, e na época em que este livro estava sendo escrito o Google havia acabado de lançar seu smartphone Nexus 6, com Android 5.0 Lollipop, tela Quad HD de 6 polegadas com 2560 x 1440px e um processador quad core de 2.7 Ghz; o mais rápido de todos os smartphones Android já lançados até o momento.

Recentemente, também foram lançados os tablets Nexus 7 e Nexus 10 do Google, com telas de 7 e 10 polegadas. No site da linha Nexus, você pode encontrar sempre os modelos mais atualizados dos smartphones e tablets oficiais do Google. Uma das vantagens de ter um smartphone Nexus é porque eles são chamados de Android puros, ou seja, não contêm customizações. E por serem gerenciados pelo Google sempre recebem a atualização de novas versões do sistema operacional de forma rápida. Para mais informações sobre a linha Nexus, e uma ótima explicação dos recursos do Android, visite o site:

<http://www.google.com.br/nexus/>.

1.12 Um pouco sobre a história e versões do Android

A versão 1.0 do Android chegou no mercado em 2008, com o famoso T-Mobile G1, e depois o Android não parou mais de evoluir.

Algo interessante e engraçado sobre o Android é que cada nova versão é apelidada carinhosamente com o nome de um doce. Isso gera sempre uma grande expectativa e especulação no mercado, pois todos ficam tentando adivinhar qual será o novo sabor do Android.

Na 1ª edição deste livro, expliquei o que era o Android, mas agora a história é um pouco diferente, pois o Android é o sistema operacional móvel mais utilizado no

mundo; por essa razão, acho conveniente explicar um pouco de sua história, e o que cada versão trouxe de novidades para a plataforma.

1.13 Android 1.5 (Cupcake)

Lançado em abril de 2009, o Cupcake (Figura 1.1) trouxe na época diversas melhorias para o sistema operacional, como na parte de câmera, GPS, upload de fotos e vídeos para o YouTube e Picasa etc.

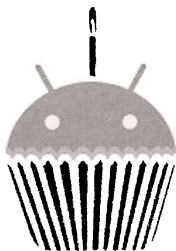


Figura 1.1 – Android 1.5 (Cupcake).

A principal novidade, porém, foi o lançamento do primeiro Android (HTC Magic) com apenas o touch screen e o teclado virtual. Foi no Cupcake que nasceram os widgets, que são miniaplicativos que podem executar na tela inicial.

Fontes:

<http://developer.android.com/about/versions/android-1.5.html>

<http://developer.android.com/about/versions/android-1.5-highlights.html>

1.14 Android 1.6 (Donut)

Lançado em setembro de 2009, o Donut (Figura 1.2) inovou e fez o Android falar e escutar.

Converter texto em voz é o que chamamos de Text-To-Speech (TTS), e o contrário, converter voz em texto, chamamos de Speech-To-Text (STT). Com o auxílio das pesquisas de voz, a home do Android ganhou mais funcionalidades e o usuário poderia pesquisar na agenda de contatos, na galeria de músicas e na web com a voz.



Figura 1.2 – Android 1.6 (Donut).

No Android 1.6 foram criadas as medidas de densidade (ldpi, mdpi, hdpi) que vamos estudar ao longo do livro, pois foi quando o Android passou a ser utilizado por dispositivos de diversas resoluções e tamanhos de tela. O Android estava chegando a um novo patamar e começando a ser amplamente utilizado e adotado pelo mercado.

Fontes:

<http://developer.android.com/about/versions/android-1.6.html>

<http://developer.android.com/about/versions/android-1.6-highlights.html>

1.15 Android 2.0 e 2.1 (Eclair)

Lançado em outubro de 2009 e depois atualizado em janeiro de 2010, o Eclair (Figura 1.3) trouxe uma interface de usuário diferenciada e adicionou os Live Wallpapers (plano de fundos animados na tela inicial).



Figura 1.3 – Android 2.1 (Eclair).

No Eclair foi lançado o suporte a múltiplas contas do Google e sincronização (junto com a API), assim como diversas melhorias em todo o sistema operacional, como nas câmeras, mapas e o suporte ao HTML5.

Fontes:

<http://developer.android.com/about/versions/android-2.0.html>

<http://developer.android.com/about/versions/android-2.0-highlights.html>

<http://developer.android.com/about/versions/android-2.1.html>

1.16 Android 2.2 (Froyo)

Lançado em maio de 2010, o Froyo (Figura 1.4) trouxe diversas melhorias de desempenho para o sistema operacional, como o compilador JIT (Just In Time) e uma engine de JavaScript mais rápida.



Figura 1.4 – Android 2.2 (Froyo).

Nessa versão foram adicionados recursos clássicos como o USB Tethering e Wi-Fi Hotspot, assim como o suporte ao Flash.

Fontes:

<http://developer.android.com/about/versions/android-2.2.html>

<http://developer.android.com/about/versions/android-2.2-highlights.html>

1.17 Android 2.3 (Gingerbread)

Lançado em dezembro de 2010, o Gingerbread (Figura 1.5) trouxe novidades na câmera, pois era possível alternar entre a câmera frontal e traseira. Tivemos melhorias na funcionalidade de copy-paste, pois era possível tocar o texto e depois arrastar para controlar a seleção.



Figura 1.5 – Android 2.3 (Gingerbread).

Segundo o Google, foi nessa versão que tivemos um grande ganho com relação ao gerenciamento da bateria e surgiu o suporte ao NFC (Near Field Communications).

Fontes:

<http://developer.android.com/about/versions/android-2.3.html>

<http://developer.android.com/about/versions/android-2.3-highlights.html>

1.18 Android 3.0 (Honeycomb)

Lançado em fevereiro de 2011, o Honeycomb (Figura 1.6) trouxe um sistema operacional totalmente focado nos tablets, com uma experiência de usuário totalmente diferenciada para telas grandes.

Com o Honeycomb, o Android deixou de ter botões físicos, e os botões de voltar e início (home) passaram a fazer parte da barra de navegação dentro da tela com touch screen.

Foi nesta versão que também foi criada a action bar, que é o padrão de navegação mais utilizado nos aplicativos para Android atualmente, e também a API de fragments, que permite criar componentes reutilizáveis de código. Ambas as APIs são fundamentais no desenvolvimento de aplicativos e por isso vamos estudá-las em detalhes neste livro.

Fontes:

<http://developer.android.com/about/versions/android-3.0.html>

<http://developer.android.com/about/versions/android-3.0-highlights.html>



Figura 1.6 – Android 3.0 (Honeycomb).

1.19 Android 4.0 (Ice Cream Sandwich)

Lançado em outubro de 2011, o Ice Cream Sandwich (Figura 1.7) unificou a plataforma de desenvolvimento entre smartphones e tablets, permitindo com que

aplicativos para smartphones fossem criados com a action bar e fragments. Com o ICS, o mesmo sistema operacional agora executava em tablets e smartphones.

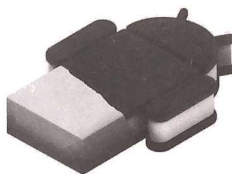


Figura 1.7 – Android 4.0 (ICS).

A API de fragments é utilizada para criar componentes reutilizáveis de código, por isso ela tem muita importância ao reaproveitar o código de um aplicativo entre as versões para tablet e smartphone.

Fontes:

<http://developer.android.com/about/versions/android-4.0.html>

<http://developer.android.com/about/versions/android-4.0-highlights.html>

1.20 Android 4.1 (Jelly Bean)

Lançado em junho de 2012, o Jelly Bean (Figura 1.8) voltou a trazer ganhos significativos com relação ao desempenho do sistema, e todo o sistema operacional ganhou melhorias no suporte às animações, deixando a interface mais sensível ao toque e fluida.



Figura 1.8 – Android 4.1 (Jelly Bean).

As notificações que são famosas no Android passaram a ser mais ricas e conter muitos detalhes.

Fonte:

<http://developer.android.com/about/versions/jelly-bean.html>

1.21 Android 4.4 (KitKat)

Lançado em outubro de 2013, o KitKat (Figura 1.9) trouxe o Android para todos, pois conseguiu executar o sistema operacional mesmo em dispositivos com menos de 512MB de RAM, devido às diversas melhorias de desempenho e otimizações feitas no sistema operacional.



Figura 1.9 – Android 4.4 (KitKat).

O KitKat trouxe aperfeiçoamentos no Bluetooth, NFC, Print Framework, sensores, e foi criada a API de Transitions, que possibilitou aos desenvolvedores, não só criarem interfaces visuais em cenas, como também animar a transição entre uma cena e outra.

Na verdade, a plataforma do Android evolui tão rápido, que para um resumo completo é recomendado olhar a documentação oficial.

Fonte:

<http://developer.android.com/about/versions/kitkat.html>

1.22 Android 5.0 (Lollipop)

Lançado em novembro de 2014, o Lollipop (Figura 1.10) foi o maior release focado na interface de usuário, usabilidade, animações e experiência do usuário.



Figura 1.10 – Android 5.0 (Lollipop).

Nasceu então o Material Design, que é um guia completo sobre como implementar o visual, animações e a interação entre os componentes de um layout, levando em consideração que o Android se tornou uma plataforma comum para vários dispositivos, como smartphones, tablets (Android), wearables (Android Wear), óculos (Google Glass), TVs (Android TV) e carros (Android Auto).

Isso é o mais importante, uma vez que as técnicas do Material Design não precisam ser implementadas somente nos smartphones e tablets, pois o Google criou um padrão de design consistente entre várias plataformas, como mobile, web, wear etc.

Dentre outras melhorias, tivemos as notificações, que agora também aparecem na tela de bloqueio (Lock Screen), e as head-up notifications, que aparecem no topo da tela com alta prioridade. Um exemplo de head-up notifications é a ligação que permite atender ou rejeitar uma ligação telefônica diretamente na notificação. Antigamente, esse recurso não existia e a aplicação da ligação mostrava uma tela cheia para o usuário decidir se atende ou não a ligação.

Outra novidade interessante foi o projeto Volta, que trouxe ferramentas para auxiliar a análise do uso da bateria nos aplicativos. Também foi modificada a tela de aplicativos recentes (Overview Screen), que mostra as últimas tarefas que estão sendo executadas, sendo que um aplicativo pode conter uma ou mais tarefas. Foi criada uma API para os desenvolvedores controlarem esse comportamento. O Lollipop também suporta a OpenGL ES 3.1, trazendo um desempenho superior nos jogos 2D e 3D.

A plataforma do Android está chegando a outro patamar, e o Google TV também recebeu um grande release. Foi criada a API Leanback para criar interfaces ricas para TV e o TIF (Android TV Input Framework).

Novamente, são tantas as novidades que recomendo olhar a documentação oficial.

Fonte: <http://developer.android.com/about/versions/lollipop.html>

1.23 Google I/O 2015 e o anúncio do Android M

No Google I/O 2015 foi anunciado o Android M, que é a prévia para desenvolvedores da nova versão do Android. A letra M dá sequência à letra L de Lollipop, e o Google está seguindo essa nomenclatura agora. Porém, oficialmente a nova versão do Android deve ser lançada entre outubro e novembro de 2015, e somente então saberemos qual é o novo sabor do Android. Como esta versão começa com a letra M, algumas das suspeitas são os famosos doces M&M's, Mentos, dentre outros.

Dentre as melhorias já anunciadas no Android M, temos um leitor de impressão digital com sensor biométrico e o Android Pay, uma plataforma aberta para fazer pagamentos por meio de cartões, internet e NFC.

Uma das funcionalidades que achei mais interessante é o novo sistema de controle de permissões dos aplicativos, o qual possibilitará que o usuário conceda a permissão individualmente. Será possível optar por dar acesso à câmera enquanto nega-se a permissão para acessar o GPS, por exemplo.

Outra funcionalidade do Android M muito comentada são os App Links, que permitem que determinados aplicativos sejam escolhidos como padrão ao abrir links de determinado domínio. Exemplos clássicos são os apps do Twitter, Google + e Drive, e nesse caso qualquer link desses domínios pode ser aberto diretamente em seus respectivos aplicativos sem a necessidade de perguntar ao usuário qual aplicativo deve ser escolhido.

O Google Now também recebeu atualizações e agora ele pode ser chamado diretamente da tela de bloqueio, pela opção no canto esquerdo inferior da tela. Enfim, agora é esperar pela nova versão do Android. No entanto, como desenvolvedor, você vai perceber que no Android SDK é possível baixar a versão prévia do Android M e já ir brincando com o emulador.

Neste livro, vamos aprender a criar aplicativos para Android. Na maioria das vezes, vou tentar manter a compatibilidade com as versões antigas, até porque o que iremos estudar são os conceitos principais do Android.

Felizmente o Google vem fazendo um excelente trabalho no suporte à compatibilidade com versões antigas, e podemos utilizar novas funcionalidades por meio de bibliotecas de compatibilidade. Um exemplo disso é a biblioteca de compatibilidade v7, que traz ao Android 2.1 (Eclair) a funcionalidade da action bar que foi criada apenas no Android 3.0 (Honeycomb).

No livro, serão exploradas diversas APIs de desenvolvimento, do básico ao avançado. Vamos focar boa parte em boas práticas de programação e interface de usuário, seguindo sempre as recomendações (guidelines) do Google. Para isso, será desenvolvido, passo a passo durante a leitura, o aplicativo dos carros, explorando muitos conceitos do Material Design.

Tenho certeza de que você, ao ler este livro, vai adquirir uma base sólida referente a todos os conceitos do Android, desde o básico ao avançado. Naturalmente, a plataforma não para de evoluir, mas estou certo de que no final da leitura você estará apto a acompanhar essa evolução.



CAPÍTULO 2

Configuração do ambiente de desenvolvimento

Para iniciar o desenvolvimento de aplicações para o Android, é necessário instalar o SDK, que contém o emulador e todas as ferramentas necessárias para o desenvolvimento.

Este capítulo aborda a instalação do SDK e a configuração do ambiente de desenvolvimento no Android Studio. Para validar a configuração do ambiente, criaremos um simples projeto para testar o emulador.

2.1 Android SDK

O Android SDK é o software utilizado para desenvolver aplicações no Android, que tem um emulador para simular o dispositivo, ferramentas utilitárias e uma API completa para a linguagem Java, com todas as classes necessárias para desenvolver as aplicações.

O Android SDK pode ser encontrado neste endereço: <http://developer.android.com/sdk/>.

Neste livro vamos utilizar o Android Studio para desenvolver aplicações para Android. Não se preocupe com o SDK neste momento, pois vamos baixar o Android Studio e ele já contém o SDK.

2.2 Requisitos de software e sistema

O Android Studio e SDK são suportados nos seguintes sistemas operacionais:

- Windows XP, Vista, 7 ou 8 (32 ou 64-bit)
- Mac OS X 10.8.5 ou posterior (somente x86)
- Linux (testado no Linux Ubuntu)

A seguir, veja as informações sobre os ambientes de desenvolvimento suportados:

- No mínimo 4GB de memória recomendável. Minha experiência é de que é necessário no mínimo 8GB de memória.
- Pelo menos 1GB livre no disco para instalar o Android SDK, emulador, imagens de sistema, ferramentas e cache.
- JDK 7 (apenas a JRE não é o suficiente). Caso a máquina já tenha um JDK inferior instalado, como por exemplo o JDK 1.4, verifique se as configurações da variável de ambiente `PATH` e `JAVA_HOME` do sistema operacional estão utilizando as versões corretas.

Nota: ao instalar o Java, configure a variável de ambiente `JAVA_HOME` do sistema operacional para apontar para a pasta na qual o Java JDK está instalado.

2.3 Plataforma (versão do Android)

Antes de começarmos a brincar com o Android, é importante entendermos o que é API Level. Como existem vários dispositivos com Android no mercado, é possível que cada um deles tenha uma versão diferente do sistema operacional. Por exemplo, o primeiro smartphone Android, o HTC G1, tinha a versão 1.1, e os novos smartphones estão saindo com as versões mais recentes.

No Android, uma versão do sistema operacional é conhecida como plataforma. Podemos dizer então que existem diversas plataformas diferentes do Android (1.1, 1.5, 1.6, 2.x, 3.x, 4.x, 5.x etc.).

Cada plataforma tem um código identificador, chamado de API Level. A lista a seguir mostra a relação entre o código API Level e cada plataforma:

- **API Level 1** – Corresponde à plataforma do Android 1.0.
- **API Level 2** – Corresponde à plataforma do Android 1.1.
- **API Level 3** – Corresponde à plataforma do Android 1.5 (Cupcake).
- **API Level 4** – Corresponde à plataforma do Android 1.6 (Donut).
- **API Level 5** – Corresponde à plataforma do Android 2.0.
- **API Level 6** – Corresponde à plataforma do Android 2.0.1.
- **API Level 7** – Corresponde à plataforma do Android 2.1 (Eclair).

- **API Level 8** – Corresponde à plataforma do Android 2.2 (Froyo).
- **API Level 9** – Corresponde à plataforma do Android 2.3 (Gingerbread).
- **API Level 10** – Corresponde à plataforma do Android 2.3.3.
- **API Level 11** – Corresponde à plataforma do Android 3.0 (Honeycomb).
- **API Level 12** – Corresponde à plataforma do Android 3.1.
- **API Level 13** – Corresponde à plataforma do Android 3.2.
- **API Level 14** – Corresponde à plataforma do Android 4.0 (Ice Cream Sandwich).
- **API Level 15** – Corresponde à plataforma do Android 4.0.3.
- **API Level 16** – Corresponde à plataforma do Android 4.1 (Jelly Bean).
- **API Level 17** – Corresponde à plataforma do Android 4.2 (Jelly Bean).
- **API Level 18** – Corresponde à plataforma do Android 4.3 (Jelly Bean).
- **API Level 19** – Corresponde à plataforma do Android 4.4 (KitKat).
- **API Level 20** – Corresponde à plataforma do Android 4.4W (KitKat para wearables).
- **API Level 21** – Corresponde à plataforma do Android 5.0 (Lollipop).
- **API Level 22** – Corresponde à plataforma do Android 5.1 (Lollipop MR1).
- **API Level X** – Novas versões do Android vão continuar a contagem...

A API Level é um número identificador valioso para aplicações Android, uma vez que, ao desenvolver aplicações, é necessário definir quais serão os dispositivos alvos. Dessa forma, se você sabe que utilizará uma nova API que existe apenas em smartphones Android 4.x ou superior, será necessário definir no projeto que a API Level mínima suportada é a 14 – compatível com o Android 4.0 ICS.

Para obter a lista atualizada de API Level, visite este site:

<http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#ApiLevels>

2.4 Android Studio

Neste livro vamos adotar o Android Studio, que é a IDE oficial de desenvolvimento para Android. O Android Studio foi anunciado no Google I/O 2013 e é baseado no IntelliJ IDEA da JetBrains.

O Android Studio apresenta alguns diferenciais importantes se comparado ao Eclipse, que antigamente era a ferramenta oficial.

1. Editor visual mais fluido e com mais opções.
2. Sistema de build mais moderno baseado em Gradle (gradle.org).
3. Diversas utilidades e facilidades ao desenvolver para Android, sendo muito integrado ao Android SDK.
4. Templates de projetos para smartphones, tablets, relógios etc.
5. Atualizações e melhorias frequentes.

Uma grande diferença entre o Eclipse e o Android Studio é o processo de compilação dos projetos. No Eclipse cada projeto é compilado do jeito clássico, como qualquer projeto Java dentro do Eclipse. Mas no Android Studio a compilação é feita pelo Gradle, que é um moderno sistema de builds. Segundo o site oficial do Gradle (gradle.org), ele é definido com a seguinte frase: “Gradle combina o poder e a flexibilidade do Ant com o gerenciamento de dependência e convenções do Maven, em uma maneira mais eficaz”.

Se você não está acostumado com gerenciamento de dependências, pode achar o Gradle complicado no início, mas fique tranquilo que durante a leitura do livro vamos praticar bastante. Com a prática do dia a dia, você vai se acostumar com ele e aos poucos vai perceber suas vantagens.

Agora vamos colocar a mão na massa e baixar o Android Studio; acesse a seguinte página, conforme a figura 2.1.

<http://developer.android.com/sdk/>

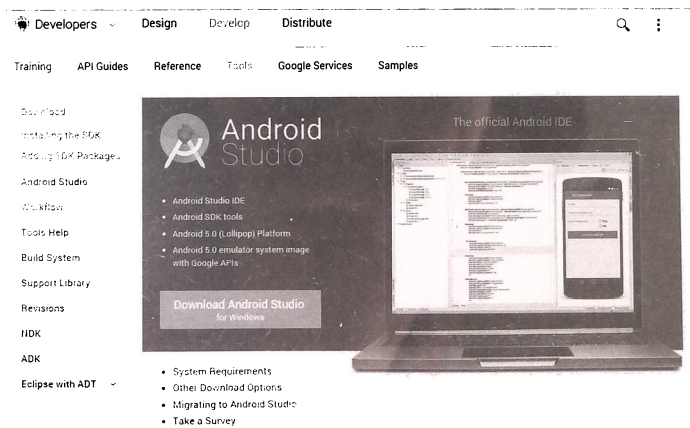


Figura 2.1 – Download do Android Studio.

Durante a instalação defina a pasta na qual o Android Studio e o Android SDK serão instalados. No wizard de instalação é recomendado deixar o Android SDK na pasta do usuário para não termos problemas de permissão ao baixar atualizações no SDK. Depois de instalar o Android Studio, a estrutura de pastas é como a exibida na figura 2.2. Na parte esquerda da figura temos a pasta de instalação do Android Studio e na direita, onde o Android SDK foi instalado.

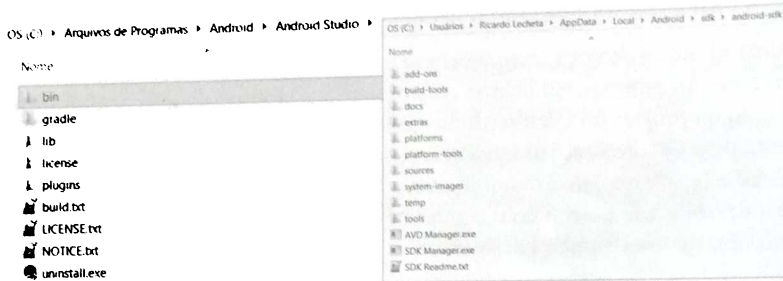


Figura 2.2 – Pasta de instalação Android Studio.

Ao executar o Android Studio você verá um wizard inicial (Figura 2.3). Na época em que este livro estava sendo escrito o Android Studio estava na versão 1.0.1, mas você verá que são lançadas novas versões com frequência, pois a ferramenta está em constante evolução e sempre recebe melhorias. A vantagem é que o processo de atualização é automático e você receberá uma alerta sempre que existir uma atualização.

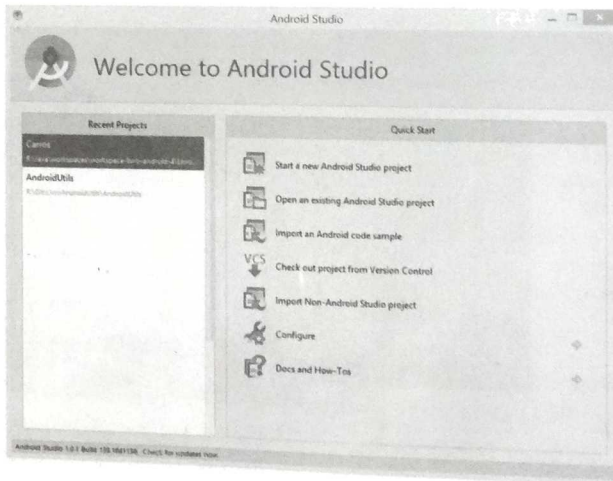


Figura 2.3 – Primeira execução do Android Studio.

Antes de criar um projeto, recomendo clicar no botão **Configure** do wizard para fazer o download e a instalação das plataformas (versões) do Android, embora o Android Studio já venha com vários itens pré-instalados. O wizard com as opções de configurações pode ser visto na figura 2.4. Nessa página do wizard, clique no link **SDK Manager** para abrir o utilitário de instalação do Android SDK, a explicação de como continuar está no próximo tópico.

Nota: o Android Studio contém o Android SDK, mas é recomendado atualizá-lo utilizando o SDK Manager. Outra maneira de abrir o SDK Manager é pelo menu **Tools > Android > SDK Manager** caso o Android Studio esteja aberto.

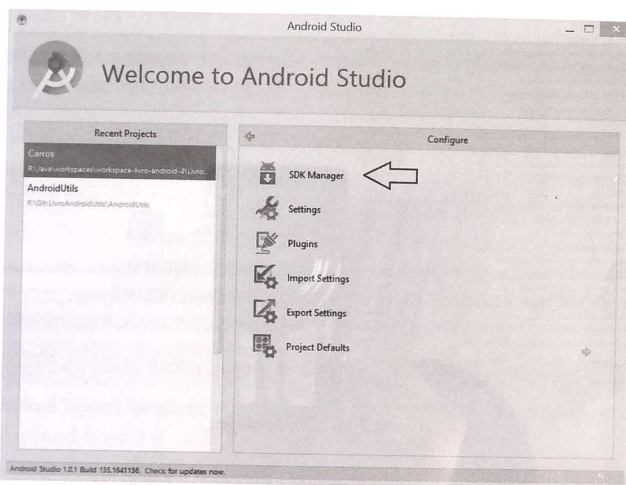


Figura 2.4 – Configurações.

2.5 Instalando os pacotes pelo SDK Manager

Para iniciar o desenvolvimento, é necessário baixar as plataformas do Android, com o objetivo de criar os emuladores para cada versão do sistema operacional.

Essa instalação é feita pelo SDK Manager. Aqui podemos baixar todas as plataformas do Android e suas respectivas documentações, o driver USB do Google para conectar um dispositivo na USB, as bibliotecas de compatibilidade, biblioteca do Google Play Services, o acelerador de emulador da Intel (HAXM) etc.

A figura 2.5 mostra o utilitário de configuração do SDK aberto e com a plataforma do Android 5.0.1 instalada. Nessa figura estou mostrando apenas os itens instalados no meu computador para você ter uma base, mas verá que vão existir vários outros itens no SDK disponíveis para instalação.

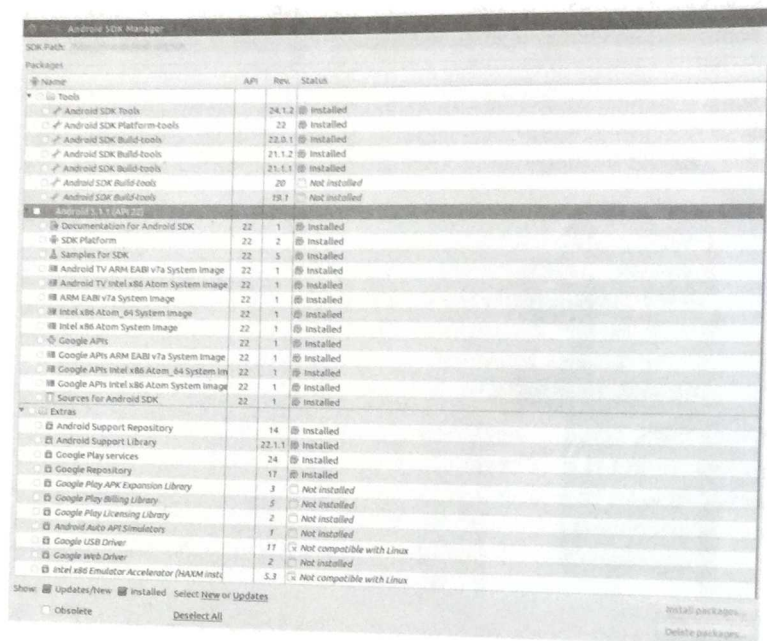


Figura 2.5 – SDK Manager.

Nota: quando você for fazer a instalação, baixe a última versão que estiver disponível. Os emuladores ARM são mais lentos. Os emuladores Intel x86 utilizam o acelerador da Intel (ver próximo tópico) e são mais rápidos.

Instalar versões da plataforma do Android e bibliotecas pelo SDK Manager é simples: basta selecionar os pacotes desejados e clicar em **Install packages**. O importante é você entender o que precisa estar instalado.

Para começar, é sempre importante manter os três primeiros itens atualizados, que são referentes ao SDK Tools, pois isso influencia diretamente na compilação do código.

- **Android SDK Tools** – Ferramentas do SDK, como o emulador.
- **Android SDK Platform-tools** – Ferramentas da plataforma do Android.

- **Android SDK Build-tools** – Ferramenta de compilação. Ele é extremamente importante, pois a versão que você baixar aqui será utilizada para compilar o projeto com o Gradle. No arquivo *build.gradle* do projeto é especificado o código da versão do build-tools que é utilizado para fazer a compilação.

Na figura 2.5 podemos ver que, abaixo do item 5.1.1 (API Level 22), foram instalados vários componentes:

- **Documentation for Android SDK** – Documentação do SDK.
- **SDK Platform** – Esse é o item mais importante, pois é a plataforma dessa versão do Android. No diretório do SDK ele será instalado na pasta */sdk/platforms*, a qual contém as classes e APIs dessa versão do Android.
- **Samples for SDK** – Documentação do SDK.
- **EABI v7a System Image** – Imagem do emulador do Android. Você pode baixar a versão da Intel x86 ou ARM.
- **Android TV System Image** – Imagem para criar o emulador do Android TV. Você pode baixar a versão da Intel x86 ou ARM.
- **Android Wear System Image** – Imagem para criar o emulador do Android Wear. Você pode baixar a versão da Intel ou ARM.
- **Google APIs System Image** – Esse item é idêntico à imagem do emulador convencional, mas ainda contém as APIs do Google. Recomenda-se sempre criar o emulador com o Google APIs.

Mais abaixo na pasta *Extras* deixei instaladas as seguintes bibliotecas.

- **Android Support Repository** – Repositório utilizado pelo sistema de build do Android Studio (Gradle).
- **Android Support Library** – Biblioteca de compatibilidade com suporte às versões mais antigas do Android. Contém várias classes que permitem criar aplicativos que funcionem de forma coerente em todas as versões do Android.
- **Google Play Services** – Bibliotecas adicionais do Google como Mapas V2, Localização, Google Fit, Google Drive, GCM (Push) etc.
- **Google Repository** – Repositório interno utilizado pelo Google.
- **Google USB Driver** – Driver para os smartphones e tablets da linha Nexus do Google. Esse item é necessário apenas no Windows.
- **Intel x86 Emulator Accelerator (HAXM installer)** – Esse item é um acelerador de velocidade do emulador para Windows. Depois de baixá-lo, é necessário instalá-lo. Com isso é possível criar os emuladores x86 que são mais rápidos.

2.6 Intel Hardware Accelerated Execution Manager (HAXM)

O emulador do Android é famoso por sua lentidão e muitos desenvolvedores optam por desenvolver diretamente com um dispositivo real plugado na USB ou instalar emuladores de terceiros, como o Genymotion (<https://www.genymotion.com/>).

Para solucionar esse problema de lentidão do emulador, a Intel criou um acelerador para o emulador. Isso é possível graças ao Intel Hardware Accelerated Execution Manager (HAXM). Com a tecnologia de virtualização, o emulador do Android consegue executar instruções a cerca de 80% da velocidade nativa do processador host, o que significa na prática um emulador cerca de cinco vezes mais rápido. O Intel HAXM pode ser baixado pelo SDK Manager e tem suporte para os principais sistemas operacionais: Windows, Mac OS e Linux. Depois de baixá-lo, você precisará entrar na pasta `/sdk/extras/intel` e instalar o software (Figura 2.6), pois é feito apenas o download do instalador.

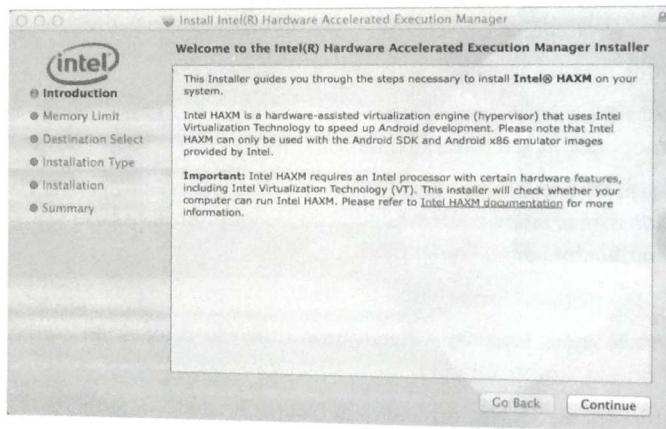


Figura 2.6 – Instalando o acelerador do emulador da Intel.

Vale lembrar que o HAXM é compatível com processadores Intel e suporta virtualização com Intel® VT-x. Caso consiga fazer a instalação com sucesso, você poderá utilizar as imagens x86 do emulador do Android, Android Wear e Google TV da Intel, que são os itens com o nome **Intel x86 Atom System Image** que aparecem no SDK Manager.

Esses emuladores, chamados de x86, são muito mais rápidos que a versão ARM, pois utilizam o processador e a memória do computador host para executar o emulador. Enfim, se seu computador aceitar a instalação do HAXM, é uma ótima notícia. Para mais informações sobre o HAXM visite o site da Intel:

<https://software.intel.com/pt-br/blogs/2012/06/25/decole-com-seu-emulador-android>

<https://software.intel.com/pt-br/android/articles/intel-hardware-accelerated-execution-manager>

2.7 Criando um projeto no Android Studio

Depois de instalar os pacotes do SDK e baixar a versão do Android desejada, vamos continuar e criar um projeto.

Abra o Android Studio e clique no link **New Project** do wizard. Feito isso, preencha o formulário com o nome do projeto, conforme a figura 2.7. O campo **Application Name** é o nome do projeto e o campo **Company Domain** é o domínio de sua empresa. Eu usei o site do livro como domínio. O item **Package name** é derivado dos campos nome do projeto e domínio, que nesse caso ficou *br.com.livroandroid.helloandroidstudio*. O pacote é quem identifica o aplicativo no dispositivo e precisa ser único, pois não é possível publicar dois aplicativos com o mesmo pacote no Google Play.

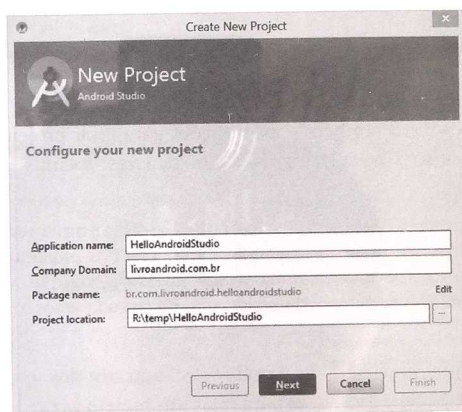


Figura 2.7 – Criando um projeto.

Por padrão, o Android Studio vai criar os projetos na pasta do usuário no sistema operacional, no subdiretório *AndroidStudioProjects*. Eu particularmente costumo mudar esse local para outra pasta.

Clique em **Next** para continuar. Na próxima página do wizard, selecione o item **Phone and Tablet** para criar um projeto compatível com smartphones e tablets (Figura 2.8), em seguida selecione no campo **Minimum SDK** a API Level 21 (Android 5.0), ou a maior versão que existir na época que você estiver lendo o livro.

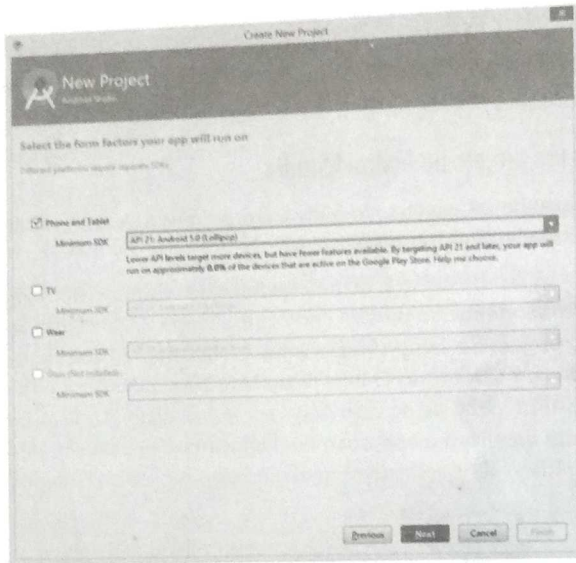


Figura 2.8 – Criando um projeto.

Observe que no wizard também temos opções para criar projetos com suporte ao Google TV, Android Wear (relógios) e Google Glass (oculos).

Importante: ao criar seu primeiro projeto Android, escolha a maior versão disponível no campo **Minimum SDK**, assim os exemplos explicados nos próximos tópicos vão funcionar. Se você selecionar alguma versão antiga do Android, o projeto será criado com a biblioteca de compatibilidade, mas isso estudaremos somente depois.

Na próxima página do wizard, você pode selecionar um dos templates disponíveis. Selecione o template **Blank Activity** para criar um projeto vazio com uma activity simples (Figura 2.9). Uma activity é uma classe que vai exibir uma tela para o usuário e controlar os eventos, sendo que esse wizard vai criar uma simples tela com uma mensagem de hello world.

Na próxima página do wizard (Figura 2.10) você deve digitar o nome da classe da activity. Digite **MainActivity** no campo **Activity Name** e os demais campos serão preenchidos automaticamente. O campo **Layout Name** é referente ao arquivo XML do layout da activity que será gerado. O campo **Title** é o título que sua aplicação vai ter ao executar no emulador. O campo **Menu Resource Name** é o XML que contera os itens de menu que serão inseridos na action bar.

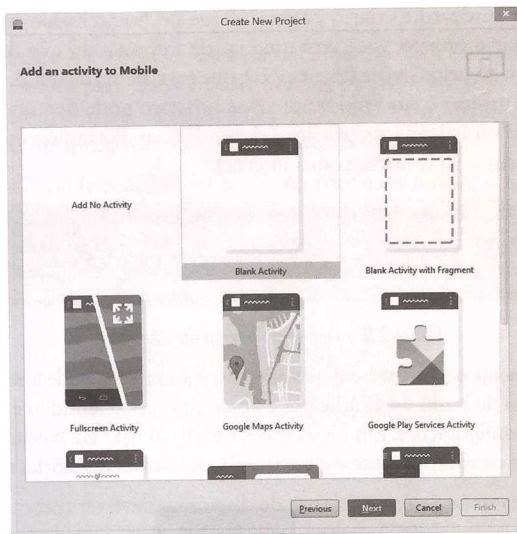


Figura 2.9 – Definindo o template do projeto.

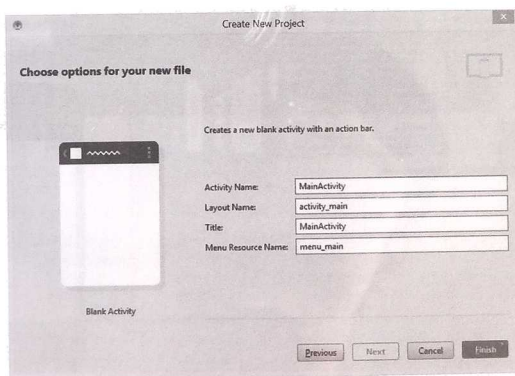


Figura 2.10 – Definindo o template do projeto.

Para os arquivos em XML, é importante seguir o padrão da conversão de nomes. Por exemplo, se sua activity se chamar `LoginActivity.java`, o arquivo XML vai se chamar `activity_login.xml`, sempre começando com a palavra `activity`, e o item de menu se chamará `menu_login.xml`. Como teremos vários tipos de XML no projeto, essa convenção facilita encontrar e identificar os arquivos.

Para finalizar o wizard e criar o projeto, clique no botão **Finish** e aguarde. Logo depois de criar o projeto, você verá uma janela informando que o projeto está sendo compilado pelo Gradle (Figura 2.11), que é o sistema de build. Na primeira vez que você utilizar o Android Studio esse processo pode demorar um pouco, pois o Gradle vai baixar seus pacotes e dependências. Portanto, certifique-se de que você possui uma conexão com a internet.

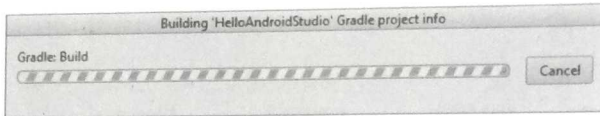


Figura 2.11 – Definindo o template do projeto.

Aqui é exatamente o momento em que um bom computador pode fazer a diferença, pois o sistema de build do Gradle pode apresentar certa lentidão em máquinas com baixas configurações. Em meus testes um disco SSD faz bastante diferença na velocidade de compilação, e é claro que, quanto mais memória, melhor.

Depois de criar o projeto, a estrutura de diretórios deve ser como a figura 2.12, que mostra o editor aberto no arquivo `/res/layout/activity_main.xml`.

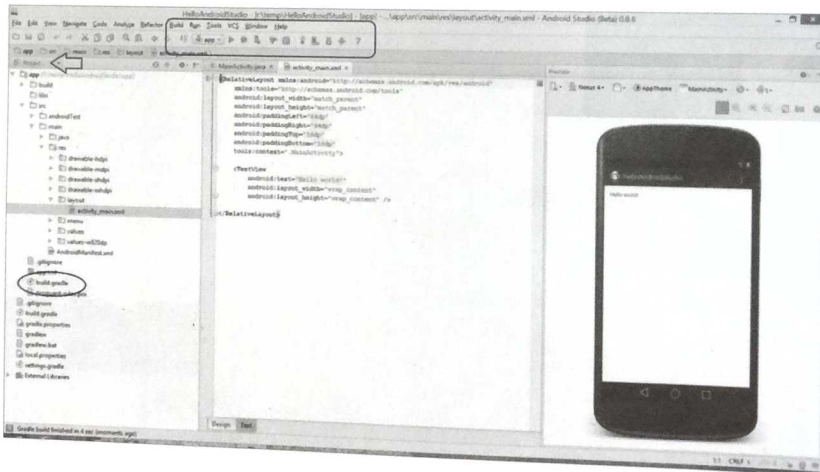


Figura 2.12 – Projeto criado e aberto no editor visual.

Na figura eu destaquei com uma seta na esquerda um item de menu que permite customizar a forma de visualização do projeto a qual está marcada como **Project**, a qual mostra a estrutura real de arquivos e pastas do projeto. Com o tempo você irá utilizar muito a visualização **Android**, que mostra de forma enxuta e agrupada os

itens do projeto. Veja que também destaquei na barra de ferramentas do Android Studio as principais opções que você irá trabalhar no dia a dia, que são o botão de **Run/Debug**, botões para abrir o SDK Manager e AVD Manager etc. Também destaquei o arquivo *app/build.gradle*, que é onde você irá configurar informações sobre a versão do aplicativo e também declarar as dependências.

Para executar o projeto, clique no botão **Run** conforme mostra a figura 2.13. Na figura o combo mostra o valor *app*, referente ao módulo *app* do projeto, pois um projeto no Android Studio é composto de vários módulos. Por exemplo, caso você crie um projeto com suporte ao Android Wear, existirá a opção para executar a versão *mobile* ou *wear* do projeto.



Figura 2.13 – Barra de ferramentas para executar o projeto.

A figura 2.14 mostra o wizard para escolher o dispositivo que deve executar o projeto. Neste caso ela está mostrando um smartphone Nexus 5 que está conectado na USB. Agora basta clicar o botão **OK** para executar o projeto direto no dispositivo.

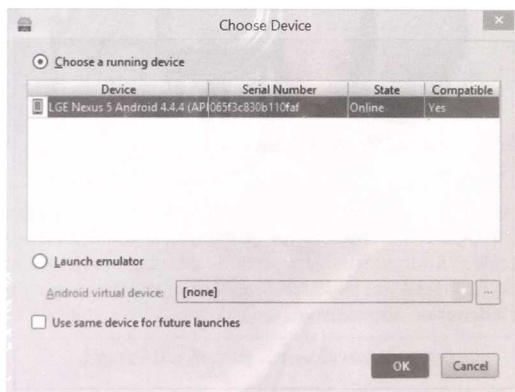


Figura 2.14 – Selecione o dispositivo ou emulador.

Dica: para depurar em um dispositivo real conectado na USB, é necessário habilitar nas configurações a opção **Security > Unknown Sources** (Segurança > Fontes desconhecidas) e a opção **Developer Options > USB Debugging** (Opções do desenvolvedor > Depuração USB). Vale alertar que o menu **Developer Options** não aparece por

padrão no Android 4.2 ou superior. Para habilitá-lo, é preciso selecionar a opção **About phone** nas configurações e clicar sete vezes seguidas na opção **Build Number**. Ao fazer isso, você receberá uma mensagem informando que agora você é um desenvolvedor e o menu **Developer Options** estará habilitado.

O resultado pode ser visto na figura 2.15, com a mensagem *Hello World*. Nesse momento eu executei o projeto diretamente no dispositivo conectado na USB, mas no próximo tópico vamos aprender a criar e instalar a aplicação no emulador.

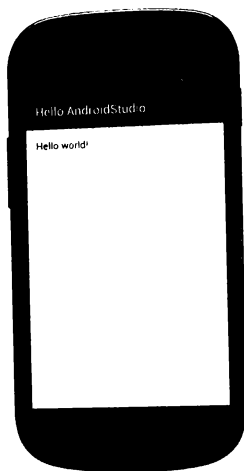


Figura 2.15 – Projeto executando no dispositivo.

Nota: para executar o projeto diretamente no dispositivo, basta conectá-lo na USB e ativar a opção USB Debugging nas configurações do dispositivo. Se o driver for reconhecido, o Android Studio vai permitir executar o projeto normalmente. Dependendo do fabricante, pode ser necessário instalar algum software para reconhecer o driver do dispositivo.

2.8 Criando um emulador (AVD)

No Android o emulador é chamado de Android Virtual Device (AVD), ou simplesmente configuração virtual de um dispositivo. O emulador simula a configuração de um smartphone ou tablet Android com exatamente a mesma plataforma do sistema operacional, resolução de tela e outras configurações.

Para criar um emulador, execute o aplicativo **AVD Manager** pelo menu **Tools > Android > AVD Manager** conforme a figura 2.16.

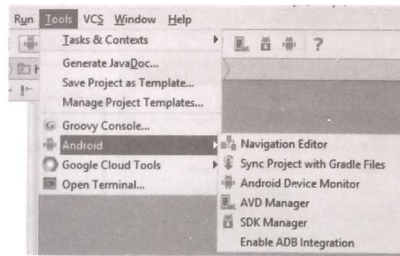


Figura 2.16 – Abrindo o AVD Manager.

Na primeira vez que você abrir o **AVD Manager**, a lista de emuladores estará vazia (Figura 2.17). Para criar um novo emulador, clique no botão **Create a virtual device**.



Figura 2.17 – Android Virtual Device Manager.

Na primeira página do wizard (Figura 2.18), selecione o tipo do dispositivo para criar o emulador, que pode ser Phone, Tablet, Wear ou TV. Veja que você pode escolher o tipo do dispositivo e o tamanho de tela que deseja simular; por exemplo, eu escolhi um Nexus S com a tela de 480x800.

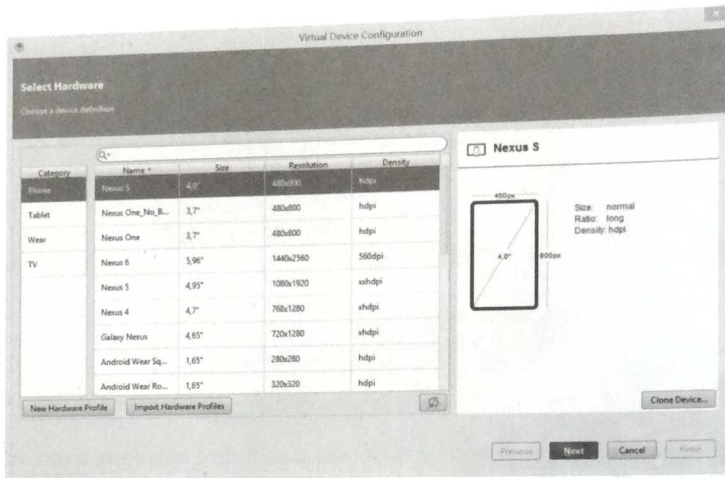


Figura 2.18 – Selecionando o tipo do dispositivo.

Na próxima página do wizard, selecione a imagem de sistema para criar o emulador do Android. Somente serão exibidas na lista as imagens de que você fez o download pelo SDK Manager. A figura 2.19 mostra a imagem do Android 5.0 API Level 21 do qual fiz o download. Note que a versão com o Google APIs é idêntica à imagem padrão, mas contém as bibliotecas do Google, portanto recomenda-se criar o emulador com o Google APIs.

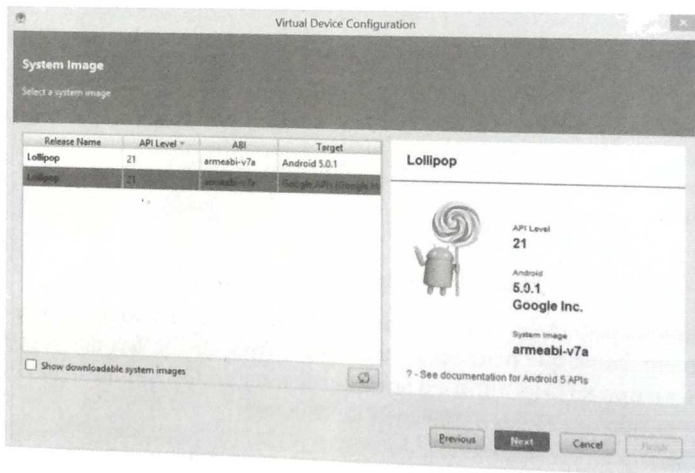


Figura 2.19 – Criando um AVD.

Dica: o emulador ARM do Android é bastante lento, por isso se puder crie o emulador x86, que tem o acelerador da Intel. Caso seu computador não suporte a tecnologia de virtualização da Intel, recomendo instalar o emulador do Genymotion (genymotion.com).

Na última página do wizard (Figura 2.20) digite um nome para o emulador e clique em **Finish**. Opcionalmente verifique as opções avançadas no item **Show Advanced Settings**.

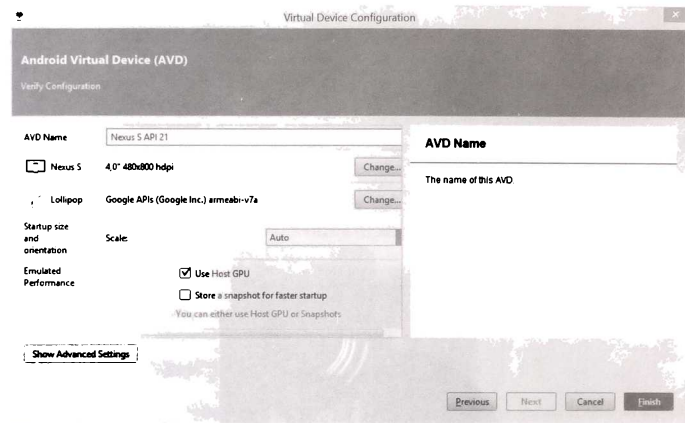


Figura 2.20 – Criando um AVD.

Depois de criar o AVD o resultado será como a figura 2.21, que mostra a lista dos emuladores criados. Para iniciar o emulador, basta selecioná-lo na lista e clicar no botão **Run** que é um triângulo verde. Nessa lista você também pode editar as configurações do emulador e até excluí-lo. Para criar mais um emulador com outra configuração, utilize o botão **Create Virtual Device**.

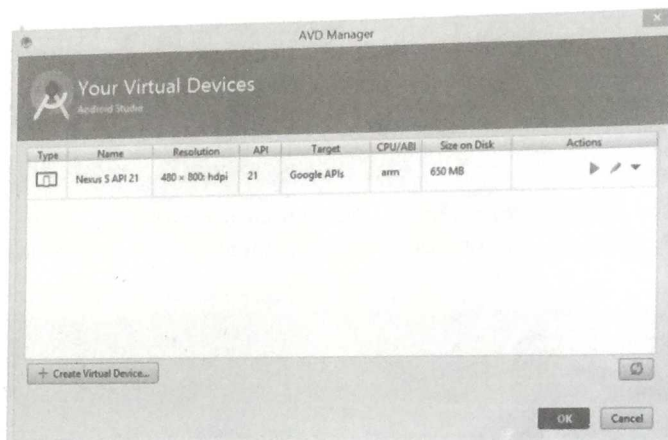


Figura 2.21 – AVD criado com sucesso.

A figura 2.22 mostra o emulador executando no emulador.



Figura 2.22 – Emulador do Android.

2.9 Executando o projeto no emulador

Ao executar o projeto, é possível instalar o aplicativo em um dispositivo conectado na USB ou em um emulador. Para executar o projeto no emulador siga os seguintes passos:

Caso o emulador esteja fechado, selecione o item **Launch emulator** (Figura 2.23) e a seguir selecione o emulador desejado. Caso o emulador esteja aberto, ele vai aparecer na lista abaixo do item **Choose a running device**.

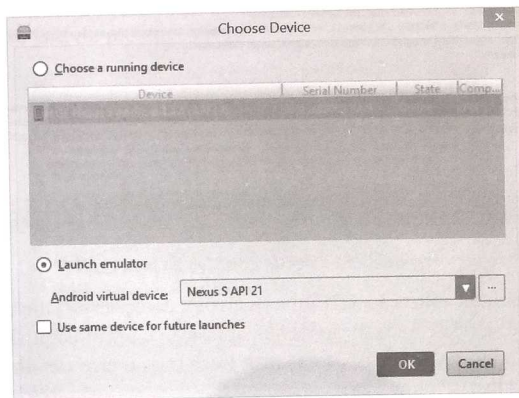


Figura 2.23 – Executando o projeto no emulador.

2.10 Algumas janelas importantes do Android Studio

Uma das coisas mais importantes ao estudar uma nova tecnologia é aprender mais detalhes sobre o ambiente de desenvolvimento, que neste caso é o Android Studio. Quanto mais familiaridade você tiver com as opções da ferramenta, melhor, mas isso vai depender muito de você, pois é só usando e fuçando que se aprende.

Uma janela importante que mostra as mensagens ao executar o projeto no emulador é a **4:Run**, conforme mostra a figura 2.24. O número 4 é do atalho para utilizar com o **Alt+Número** ou **Cmd+Número** (Mac). Na figura podemos ver os logs que mostram o aplicativo sendo instalado no emulador. É importante aprender a ler as mensagens desses logs, pois se acontecer algum erro ao executar o projeto esta janela vai mostrar detalhes importantes.

Outra janela importante é a **6:Android** (Figura 2.25), a qual mostra os logs do emulador ou dispositivo conectado na USB. No Android todos os logs são controlados pela ferramenta LogCat. Depois vamos aprender a criar esses logs dentro do código-fonte, para ajudar a depurar o código. O LogCat mostra todos os logs do sistema operacional ou apenas do processo ou tag que você especificar, o que permite depurar apenas as mensagens que lhe interessam.

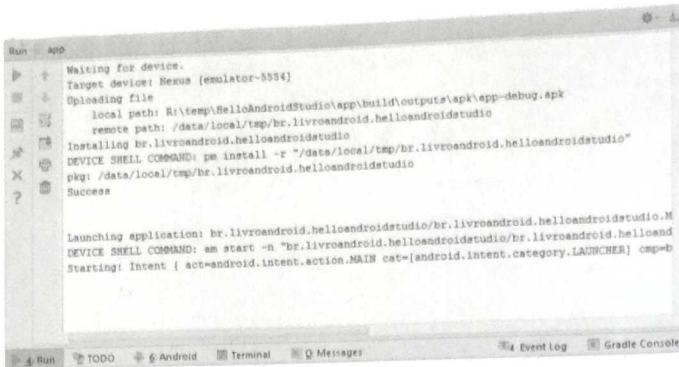


Figura 2.24 – Logs da execução do projeto.

Em caso de erro da aplicação, verifique os erros e exceções na janela do LogCat, pois tudo que se refere à execução da sua aplicação é logado aqui. Por exemplo, se o aplicativo lançar uma exceção a stack trace com o erro detalhado, ela será exibida nestes logs (Figura 2.25):

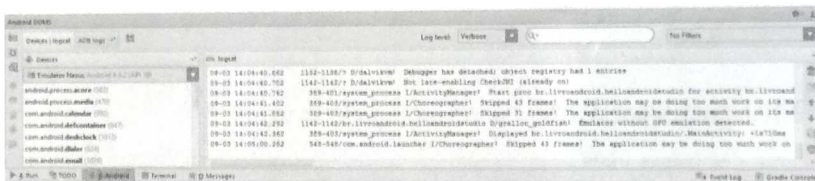


Figura 2.25 – Logs do emulador (LogCat).

Outra janela importante é a **Gradle Console**, que mostra as mensagens do build do Gradle (Figura 2.26).

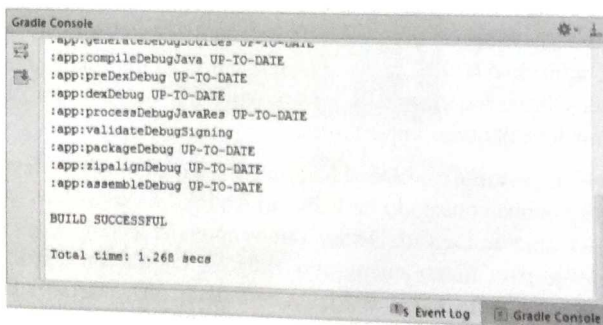


Figura 2.26 – Logs do console do Gradle.

E para turbinar o seu desenvolvimento recomendo olhar as dicas do Android Studio abrindo o wizard **Help > Tip of the Day**. Aqui você vai encontrar dicas e teclas de atalhos que vão tornar o desenvolvimento do código muito mais produtivo. A figura 2.27 mostra uma dica de atalho **Ctrl+N** que facilita encontrar e abrir qualquer classe Java do projeto.

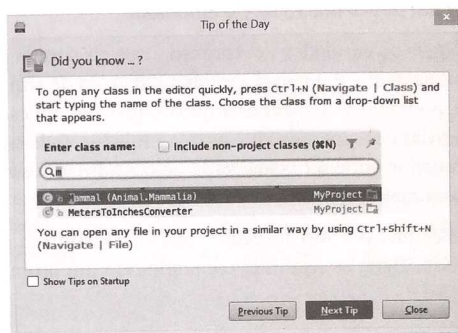


Figura 2.27 – Dicas do dia.

2.11 Aplicações na tela principal (Home)

Ao clicar no ícone **Home**, o emulador volta para a tela principal. Nessa tela, todas as aplicações instaladas podem ser visualizadas. Por padrão o ícone do aplicativo é um bonequinho do Android, conforme a figura 2.28.



Figura 2.28 – Tela Home do Android.

Esse ícone é definido no arquivo `/res/mipmap-xxx/ic_launcher.png` para cada densidade (mdpi, hdpi, xhdpi, xxhdpi etc.). Fique à vontade para alterar esse ícone e inserir uma imagem customizada para seu projeto.

2.12 Entendendo um pouco mais sobre o emulador

Uma das coisas legais do emulador do Android é que ele simula completamente o sistema operacional real, e é possível acessar a estrutura de arquivos do emulador. Dessa forma, podemos visualizar os arquivos apks das aplicações instaladas, além de excluir, enviar e recuperar arquivos do emulador. Toda essa estrutura de diretórios do emulador pode ser visualizada pelo Android Studio ou por meio de um prompt de comandos.

Demonstraremos primeiro como acessar o console do Android pelo prompt de comandos. Para isso, digite os seguintes comandos em um prompt (o emulador precisa estar aberto):

```
C:\android-studio\sdk\platform-tools>adb devices
List of devices attached
1 emulator-5554 device
```

Ao digitar o comando `adb devices` no prompt, é exibida a lista dos emuladores ativos ou dispositivos conectados, ou nada, se nenhum emulador estiver aberto no momento. Nessa lista o código (id) retornado pode ser utilizado se mais de um emulador estiver aberto para escolher em qual instância do emulador você deseja acessar o console. Nesse caso, como temos apenas um emulador aberto, basta digitar `adb shell` para acessar o console do primeiro emulador da lista.

Ao fazer isso, um prompt será aberto, permitindo navegar nos arquivos e pastas do emulador. No exemplo a seguir, foi digitado o comando `ls` do Linux para visualizar a estrutura de diretórios:

```
C:\android-studio\sdk\platform-tools>adb shell
# ls
cache
init
etc
var
data
system
tmp
root
dev
```

Observe que o sistema operacional do Android é Linux. Dessa forma, ao entrar no console do emulador, é necessário utilizar comandos do Linux, mesmo se você estiver desenvolvendo suas aplicações no Windows. As aplicações instaladas ficam na pasta `/data/app`. Vamos entrar nessa pasta para visualizar nossa primeira aplicação desenvolvida.

```
# cd /data/app
# ls
br.livroandroid.helloandroidstudio-1.apk
#
```

O arquivo `br.livroandroid.helloandroidstudio-1.apk` corresponde à aplicação final do projeto `HelloAndroidStudio` criado anteriormente. Outra forma de visualizar a estrutura de diretórios do emulador é abrir a ferramenta **Android Device Monitor** pelo menu **Tools > Android > Android Device Monitor** do Android Studio. Nessa ferramenta a janela **File Explorer** mostra a estrutura de diretórios do emulador, conforme mostra a figura 2.29.

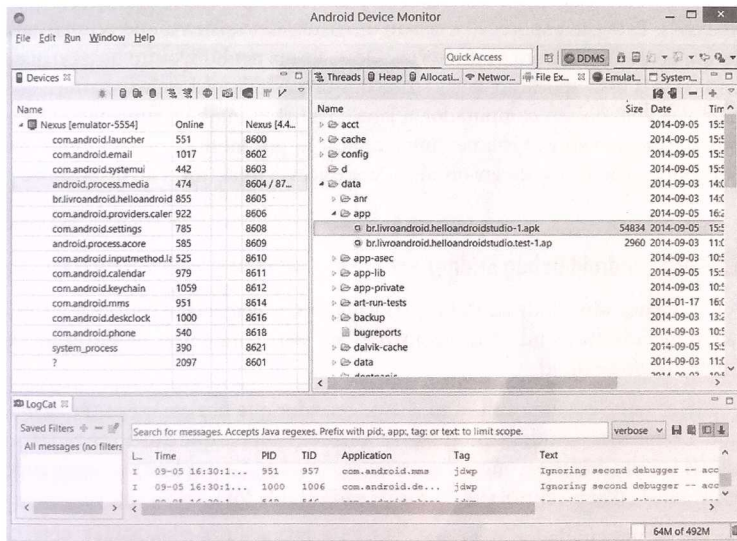


Figura 2.29 – Android Device Monitor – File Explorer.

No **Android Device Monitor** também podemos visualizar os logs Android, consumo de memória, threads etc. Uma das janelas mais interessantes é a **Devices**, que mostra todos os dispositivos conectados na USB ou até mesmo o emulador. Nessa janela podemos ver os processos que estão executando no Android, depurá-lo, matá-lo,

e inclusive tirar um screenshot da tela. Vale lembrar que as janelas **File Explorer**, **LogCat** (logs) e as outras vão mostrar as informações do dispositivo selecionado na janela **Devices**, caso exista mais de um.

Dica: no Windows tive problemas para executar o Android Device Monitor; somente funcionou ao executar o Android Studio com permissão de administrador. Essa ferramenta também pode ser executada pelo script `C:\android-studio\jdk\tools\monitor.bat`.

Para remover uma aplicação pelo **File Explorer** basta selecionar o arquivo `.apk` desejado e clicar no ícone com um traço “-” vermelho no canto direito superior da janela. Observe que nessa janela também existem ícones para enviar arquivos para o emulador e para baixar os arquivos que estão no emulador para seu computador.

Note que toda a segurança do Android é baseada na segurança do Linux. No Android cada aplicação é executada em um único processo, com uma thread dedicada. Para cada aplicação é criado um usuário no sistema operacional para ter acesso a sua estrutura de diretórios. Dessa forma nenhum outro usuário pode ter acesso a essa aplicação. Por isso, se um dispositivo real estivesse conectado na porta USB do seu computador, a janela **File Explorer** também funcionaria, mas provavelmente não exibiria nenhuma aplicação por motivos de segurança. No emulador é como se o desenvolvedor tivesse acesso total (root) ao dispositivo.

2.13 ADB (Android Debug Bridge)

A ferramenta `adb` (Android Debug Bridge) permite gerenciar e controlar o emulador. Inclusive já estudamos o comando `adb shell` para entrar na estrutura de diretórios do Android.

O `adb` consiste em uma aplicação cliente-servidor que fica em execução na máquina. Ao iniciar o Android Studio e o emulador, observe que existe um processo `adb.exe` em sua máquina, que é justamente o servidor. Esse aplicativo executando em segundo plano controla as portas de cada emulador.

Ao iniciar um emulador, sua porta-padrão é 5554 e a porta de debug, 5555. Se mais de um emulador for aberto, os próximos emuladores usarão as próximas portas sequencialmente. Por exemplo: 5556/5557, 5558/5559 etc. Com o comando `adb` é possível acessar a estrutura de diretórios do emulador da mesma forma que usando a janela **File Explorer**. Para isso, abra um prompt e digite o comando `adb shell`, e depois o `ls` para visualizar as pastas.

```
C:\android-studio\sdk\platform-tools>adb shell
# ls
. . . // você vai ver as pastas aqui
sdcard
. . .
```

Feito isso, é possível navegar na estrutura de diretórios do emulador normalmente. Veja nos comandos a seguir onde acessamos a pasta `/sdcard` para visualizar os arquivos do cartão de memória.

```
# cd sdcard
# ls
linkin_park1.mp3 // Exemplos de arquivos
linkin_park2.mp3
linkin_park3.mp3
# exit
C:\>
```

Para sair do prompt, digite `exit`. Até aqui tudo foi simples, mas se existir mais de um emulador aberto? Nesse caso o `adb` vai exibir uma mensagem de erro, pois ele não sabe qual emulador deve acessar:

```
C:\android-studio\sdk\platform-tools>adb shell
error: more than one device and emulator
```

Então, é preciso ajudá-lo. Para isso, precisamos conhecer o identificador de cada emulador; digite então o comando `adb devices`.

```
C:\android-studio\sdk\platform-tools>adb devices
List of devices attached
emulator-5554 device
emulator-5556 device
```

Esse comando listará os emuladores abertos, exibindo seu identificador, que contém a palavra `emulator` mais o número da porta em que ele está aberto, por exemplo, `emulator-5554`. Saiba que a notação pode mudar conforme o emulador; portanto, o importante é você entender o conceito. Para acessar um emulador específico, podemos utilizar o comando `adb shell` novamente e informar o parâmetro `-s` com o id da instância do emulador que desejamos.

```
C:\android-studio\sdk\platform-tools>adb -s emulator-5554 shell
```

O argumento `-s` pode ser utilizado para todos os comandos da ferramenta `adb` sempre que existir mais de um emulador aberto e for necessário informar a instância correta. Por exemplo, digamos que você deseje enviar um arquivo para o emulador. Para isso, poderá utilizar o comando `adb push`, conforme mostrado a seguir:

```
C:\android-studio\sdk\platform-tools>adb -s emulator-5554 push c:\temp\arquivo.txt /data/local/tmp
0 KB/s (0 bytes in 7.000s)
```

A sintaxe do comando `adb push` é:

```
adb push [caminho_arquivo_local] [pasta_destino_emulador]
```

Da mesma forma, podemos utilizar o comando `adb pull` para fazer justamente o contrário: o download de um arquivo que está no emulador:

```
adb pull [arquivo_origem_emulador] [caminho_arquivo_local]
```

Por exemplo, para copiar o mesmo arquivo que enviamos anteriormente para o computador, basta digitar o seguinte comando:

```
C:\android-studio\sdk\platform-tools>adb -s emulator-5554 pull /data/local/tmp/arquivo.
txt c:\temp\arquivo.txt
0 KB/s (0 bytes in 7.000s)
```

Há outras opções de uso da ferramenta `adb`, mas vamos deixar isso como leitura para você:

<http://developer.android.com/tools/help/adb.html>

Gostaria apenas de dar uma dica caso você perceba que o Android Studio e o emulador não estão conversando muito bem. Às vezes, se o emulador fica muito tempo aberto e executamos uma aplicação no Android Studio, parece que a aplicação não é instalada no emulador. É como se o emulador estivesse parado. Se isso acontecer, pode ser necessário matar o processo do `adb.exe` em execução no sistema operacional. Isso pode ser feito com o comando `adb kill-server`. E para iniciar o processo do `adb` pode-se utilizar o comando `adb start-server`. Outra forma de reiniciar o processo `adb.exe` é abrir a janela **Devices** da ferramenta Android Device Monitor e clicar no item de menu **Reset adb**.

2.14 Informações básicas sobre a resolução do emulador

Os primeiros smartphones com Android tinham uma tela com resolução HVGA (320x480), mas hoje em dia existem smartphones e tablets com diversos tamanhos de tela, o que demanda alguns cuidados durante o desenvolvimento.

O emulador do Android nos permite simular exatamente o tamanho e a resolução da tela de um dispositivo real. Para realizar essa configuração, basta escolher o tamanho da tela no momento de criar um novo emulador. Você pode escolher criar um emulador com a resolução HVGA (320x480), QVGA (240x320), WVGA (480x800) etc. Mas isso você já deve ter percebido, pois no AVD Manager, ao selecionar um tipo de dispositivo, o emulador é criado com base nessas características.

Na figura 2.30 podemos visualizar como fica a resolução da tela no emulador com diferentes configurações de AVD. As figuras mostram as resoluções WVGA alta, HVGA média e QVGA baixa, respectivamente. A resolução QVGA (240x320) são dispositivos muito pequenos, como o famoso Sony Ericsson Mini. Muitas vezes pode ser complicado desenvolver para celulares com telas tão pequenas, pois é necessário customizar o projeto para ter barra de rolagem (scroll) em todas as telas e utilizar imagens menores. Mas os smartphones com telas WVGA (480x800) são maiores e grande parte dos aparelhos modernos está vindo com esse tipo de configuração ou até superior, o que permite abusar melhor dos recursos gráficos e criar uma tela bem diferenciada.



Figura 2.30 – Emulador com as resoluções WVGA, HVGA e QVGA respectivamente.

Apenas para dar uma ideia, como cada aparelho tem uma configuração de tela, podemos utilizar imagens de tamanho diferentes para cada tipo de resolução. Para isso no projeto Android dentro do Android Studio é criada uma pasta de imagem para cada resolução. Por exemplo, existe a pasta *drawable-ldpi* para celulares QVGA (240x320) de baixa resolução, a pasta *drawable-mdpi* para celulares HVGA (320x480) de média resolução, a pasta *drawable-hdpi* para alta resolução com as telas WVGA (480x800) e por aí vai. A coisa começou a ficar tão complicada que chegou o *xhdpi* (extra high) e a história continua.

Essas pastas não têm relação com o tamanho de tela, por exemplo, 240x320 ou (480x800), e sim com a densidade e resolução da tela. O tamanho da tela pode ser o mesmo, mas a quantidade de pixels que a tela consegue exibir pode ser diferente, e isso tem a ver com a resolução. Portanto, aparelhos WVGA (480x800), embora tenham telas maiores, também apresentam resoluções muito melhores, para exibir imagens e gráficos com uma ótima definição.

Se uma imagem de 100px for inserida na pasta *drawable-mdpi*, ela será exibida perfeitamente em uma tela HVGA (320×480). Mas o que acontece se ela for exibida em uma tela menor ou maior? Nesse caso a imagem será redimensionada por padrão. Caso a imagem seja redimensionada para baixo em uma tela QVGA (240×320) não temos problemas, mas para uma tela maior como WVGA (480×800) a imagem vai distorcer e perder qualidade, o que é esperado. Por isso podemos customizar as imagens em suas respectivas pastas, para evitar que o Android faça isso em tempo de execução.

Neste exemplo, para imagem de 100px, faríamos a seguinte conta:

- Para telas QVGA (240×320), seria $100\text{px} \times 0,75 = 75\text{px}$. Nesse caso podemos inserir uma imagem de 75px na pasta *drawable-ldpi*. Ou você pode deixar que o Android redimensione a imagem para baixo em tempo de execução.
- Para telas WVGA (480×800), seria $100\text{px} \times 1,5 = 150\text{px}$. Nesse caso podemos inserir uma imagem de 150px na pasta *drawable-hdpi*, pois esses modernos aparelhos conseguem exibir imagens com muito mais definição.

O nome da imagem será sempre o mesmo. Por exemplo, se o nome do arquivo for *imagem.png*, teremos várias imagens com o mesmo nome, inseridas nas suas respectivas pastas. Em tempo de execução, conforme a resolução da tela, o Android vai escolher a pasta correta, você só precisa garantir que a imagem vai estar lá. Um exemplo prático que temos disso é o ícone da aplicação, criado automaticamente pelo wizard com o nome *ic_launcher.png* e replicado nas pastas *mipmap-mdpi*, *mipmap-hdpi*, *mipmap-xhdpi* e *mipmap-xxhdpi* para customizar o tamanho para cada resolução. Isso evita que o Android faça esse redimensionamento em tempo de execução, pois os recursos já foram informados em tempo de compilação.

Vamos aprender mais sobre esses detalhes durante o livro, então fique tranquilo.

2.15 Como fazer o download dos exemplos do livro

Se você quiser fazer o cadastro no site para mantermos contato, futuramente posso avisá-lo de novos lançamentos.

<http://www.livroandroid.com.br/>

Ou mantenha contato nas redes sociais, que costumo postar sobre lançamentos.

<http://facebook.com/ricardolecheta>

<https://plus.google.com/+RicardoLecheta>

Caso queira apenas baixar o código-fonte do livro, acesse o repositório de código-fonte do GitHub nos seguintes endereços:

<http://www.github.com/livroandroid/capitulos>

<http://www.github.com/livroandroid/carros>

<http://www.github.com/livroandroid/AndroidUtils>

O repositório */capitulos* contém várias pastas separadas por capítulo. Aqui você vai encontrar os projetos prontos referentes aos exercícios de cada capítulo. O repositório */carros* contém um passo a passo que vou utilizar durante o desenvolvimento do projeto dos carros. Cada pasta corresponde a uma parte do projeto dos carros concluído. O repositório */AndroidUtils* contém um projeto do tipo biblioteca com classes utilitárias para nos auxiliar nos exemplos.

Pela explicação do livro, você conseguirá seguir os exemplos e entender o que significa cada pasta. O objetivo dessas pastas é servir de gabarito para você. Recomendo que você tente fazer sozinho todos os exercícios de cada capítulo e o projeto dos carros passo a passo, mas se em algum momento tiver problemas basta conferir o resultado com os exemplos do GitHub.

Para fazer o download do código-fonte do GitHub você pode simplesmente clicar na opção **Download Zip**. Outra opção é clonar o repositório na sua máquina, assim se eu fizer qualquer atualização, como por exemplo uma possível correção de bug ou melhoria de algum exemplo, você sempre pode atualizar o repositório.

Com o Git instalado na máquina, abra um prompt e digite o seguinte comando para clonar o repositório:

```
git clone https://github.com/livroandroid/carros.git
```

Depois para atualizar o código caso existam atualizações utilize o comando `git pull`.

Uma vez que você fez os downloads dos exemplos, utilize o menu **File > Open** do Android Studio para abrir o projeto desejado. Agora é com você, leia com calma cada capítulo e sempre confira as explicações com os projetos de exemplo, que podem servir de gabarito para você.

Vamos lá! Desejo desde já uma boa leitura.



CAPÍTULO 3

Conceitos básicos do Android

Este capítulo aborda alguns conceitos básicos do Android, como criar telas na aplicação, definir uma interface gráfica simples, tratar eventos da tela e imprimir mensagens (logs) da aplicação utilizando a ferramenta LogCat.

Um layout de tela no Android pode ser criado utilizando um arquivo XML que define os elementos da tela ou utilizando diretamente as classes da API Java. Ao final deste capítulo, você será capaz de criar telas simples na aplicação, definir eventos para os botões e ainda visualizar os logs gerados pela aplicação.

3.1 Estrutura do projeto no Android Studio

No capítulo anterior, instalamos o Android Studio e criamos o nosso primeiro projeto para Android. Neste capítulo vamos estudar mais detalhes sobre a estrutura do projeto e os conceitos básicos de desenvolvimento para Android.

O Android Studio pode abrir um projeto de cada vez, e cada projeto pode conter um ou mais módulos. A figura 3.1 mostra a estrutura do projeto que criamos com a pasta do módulo `app` fechada. A pasta `app` representa o módulo padrão que é criado no projeto. Dentro da pasta `app` temos o código-fonte e os arquivos de compilação específicos desse módulo. Já na raiz do projeto existem os outros arquivos, como por exemplo o arquivo `build.gradle` geral do projeto, que vale para todos os módulos.

Nota: a estrutura de pastas e arquivos do projeto segue a estrutura de build do Gradle.

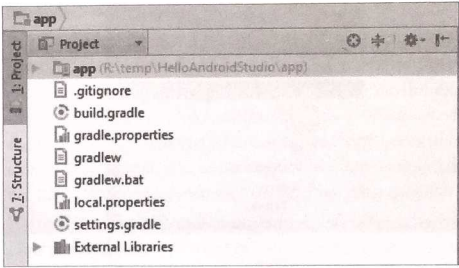


Figura 3.1 – Estrutura de arquivos do projeto.

A lista a seguir explica os arquivos que ficam na raiz do projeto.

Pasta	Descrição
<i>app</i>	Módulo <i>app</i> do projeto. O Android Studio abre um projeto de cada vez e um projeto pode conter mais de um módulo. O módulo <i>app</i> é o padrão.
<i>build.gradle</i>	Arquivo de configuração do Gradle que vale para todo o projeto, incluindo todos os módulos. Você provavelmente não vai alterar nada nesse arquivo.
<i>gradle.properties</i>	Arquivo de propriedades para customizar o build do Gradle.
<i>gradlew.bat</i>	Script que executa o build do Gradle para compilar o projeto.
<i>local.properties</i>	Arquivo com as configurações locais do projeto, como por exemplo o caminho no qual o Android SDK está instalado. Ao abrir um projeto já existente, você deve atualizar esse arquivo com o local do SDK; por sorte, o Android Studio já faz isso automaticamente para você.
<i>settings.gradle</i>	Arquivo de configuração do Gradle que indica quais módulos devem ser compilados. Se você abrir esse arquivo, verá que ele está incluindo o módulo <i>app</i> .

Ao expandir a pasta *app* você verá os arquivos referentes a esse módulo, e basicamente é aqui que ficam os arquivos da aplicação. Por exemplo, na pasta */app/src/main/* estão os arquivos *.java*, *.xml* e imagens do aplicativo, conforme a figura 3.2.

Para visualizar a estrutura de diretórios real do projeto, selecione o item **Project** no combo de opções que fica na parte esquerda superior da janela. Outra visualização muito boa é a **Android**, que mostra um resumo dos principais arquivos e deixa a visualização mais enxuta.

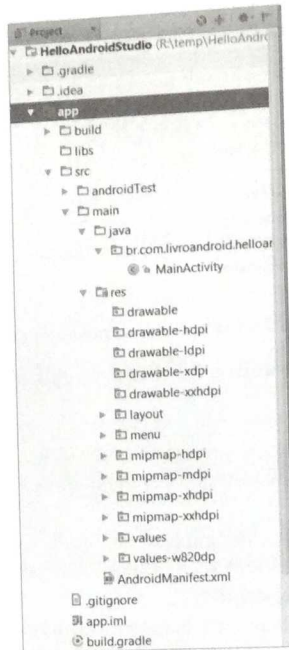


Figura 3.2 – Estrutura de arquivos do projeto.

A seguinte lista explica os arquivos do módulo `app`.

Pasta	Descrição
<code>build</code>	Pasta em que ficam os arquivos compilados do módulo. O arquivo apk que é o aplicativo compilado fica na pasta <code>build/outputs/apk</code> .
<code>R.java</code>	A classe <code>R.java</code> é gerada automaticamente ao compilar o projeto e permite que a aplicação acesse qualquer recurso como arquivos e imagens utilizando as constantes desta classe. Essa classe nunca deve ser alterada manualmente. O arquivo <code>R.java</code> é gerado na pasta <code>app/build/generated/source/r</code> do módulo.
<code>libs</code>	Pasta para inserir os arquivos <code>.jars</code> que devem ser compilados com o projeto.
<code>src/main/java</code>	Pasta com as classes Java; por exemplo, a classe <code>MainActivity</code> que foi criada pelo wizard.
<code>src/main/res</code>	Pasta que contém os recursos da aplicação, como imagens, layouts de telas e arquivos de internacionalização. Existem cinco subpastas: <code>drawable</code> , <code>layout</code> , <code>menu</code> , <code>mipmap</code> e <code>values</code> , com os seus respectivos classificadores, que vamos verificar mais adiante.

Pasta	Descrição
<i>res/drawable</i>	Pasta com as imagens da aplicação. Atualmente como existem diversos dispositivos Android com diferentes resoluções de telas, é possível customizar as imagens para ficar com o tamanho exato em cada resolução. Para isso, há diversas pastas para as imagens: <i>drawable-ldpi</i> (low), <i>drawable-mdpi</i> (medium), <i>drawable-hdpi</i> (high), <i>drawable-xdpi</i> (extra high) e <i>drawable-xxdpi</i> (extra extra high). Na última versão do Android Studio ele não cria todas as variações, eu criei para mostrar a você.
<i>res/mipmap</i>	Pasta com o ícone da aplicação, o qual por padrão chama-se <i>ic_launcher.png</i> . Da mesma forma que a pasta <i>/res/drawable</i> , esta pasta também apresenta variações conforme a densidade da tela do dispositivo.
<i>res/layout</i>	Pasta que contém os arquivos XML de layouts para construir as telas da aplicação.
<i>res/menu</i>	Pasta que contém os arquivos XML que criam os menus da aplicação, que são os botões com ações na action bar.
<i>res/values</i>	Pasta que contém os arquivos XML utilizados para a internacionalização, configuração de temas e outras configurações.

Cada arquivo seja imagem ou XML dentro da pasta */app/res* contém uma referência na classe *R*, que é automaticamente gerada ao compilar o projeto. Cada vez que se altera, adiciona ou remove um arquivo dessas pastas, a classe *R* é alterada para refletir as informações.

Por exemplo, se você inserir uma imagem chamada *smile.png* em uma das pastas */res/drawable*, a constante *R.drawable.smile* será automaticamente gerada na classe *R*. Se você inserir um arquivo XML chamado *teste.xml* na pasta */res/layout*, a constante *R.layout.teste* será gerada. A classe *R* é nossa grande aliada no desenvolvimento para Android e nunca deve ser alterada manualmente. Observe que o código-fonte da classe *MainActivity* utiliza a constante *R.layout.activity_main*, a qual define o layout utilizado para apresentar a tela ao usuário.



MainActivity.java

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main); // O layout da tela é definido aqui
    }
    @Override
```

```

    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.activity_main, menu); // Menu com botões na action bar
        return true;
    }
}

```

Nota: no código fonte da classe `MainActivity`, se você segurar o `Ctrl` e clicar em cima do código `R.layout.activity_main`, o Android Studio vai abrir automaticamente o arquivo de layout `/res/layout/activity_main.xml` no editor visual. Essa técnica com o `Ctrl+clicque` é utilizada em muitos outros lugares.

3.2 Arquivo `AndroidManifest.xml`

O arquivo `AndroidManifest.xml` é a base de uma aplicação Android e contém todas as configurações necessárias para executar a aplicação.

Quando criamos o projeto `HelloAndroidStudio` no capítulo anterior, foi criada a `MainActivity` e ela foi configurada automaticamente no arquivo `AndroidManifest.xml` como a activity inicial do projeto, ou seja, é aquela activity que contém o ícone na tela inicial (Home) e representa o aplicativo que o usuário vai executar. No código a seguir, fiz comentários nas partes mais importantes do arquivo `AndroidManifest.xml`, portanto leia com atenção.

`AndroidManifest.xml`

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="br.com.livroandroid.helloandroidstudio"> // Pacote único do projeto
    <application
        android:allowBackup="true" // Se for true então permite backup no cloud
        android:icon="@mipmap/ic_launcher" // Ícone do aplicativo na tela Home
        android:label="@string/app_name" // Nome do aplicativo na tela Home
        android:theme="@style/AppTheme" > // Tema do aplicativo (/res/values/styles.xml)
        <activity
            android:name=".MainActivity" // Classe da Activity que deve ser executada.
            android:label="@string/app_name"> // Título da activity para mostrar na action bar
            <intent-filter> // Declara um filtro para executar a activity
                // A ação MAIN indica que esta activity pode ser executada como a inicial
                <action android:name="android.intent.action.MAIN" />
            </intent-filter>
        </activity>
    </application>
</manifest>

```

```

// A categoria LAUNCHER indica que o ícone da activity deve ficar disponível
// na tela inicial
<category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
</activity>
<!-- Se o aplicativo tiver outras activities, precisa declarar aqui, segue exemplo: -->
<activity android:name=".LoginActivity" />
<activity android:name=".CadastroUsuarioActivity" />
<activity android:name=".BemVindoActivity" />
</application>
</manifest>

```

O XML sempre deve iniciar com as linhas a seguir, declarando o namespace para validar os atributos utilizados e o pacote da aplicação. O encode do XML deve ser UTF-8.

```

<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="br.com.livroandroid.helloandroidstudio">

```

Dentro da tag <manifest> é declarado o pacote principal do projeto, utilizando a tag <package>. Esse é o pacote identificador do projeto e deve ser único no Google Play. O mesmo pacote é declarado no arquivo *app/build.gradle*, o qual vamos verificar mais tarde.

É obrigatório que cada activity do projeto esteja declarada no arquivo *AndroidManifest.xml* e para isso utilizamos a tag <activity>, a qual recebe o nome da classe.

Geralmente um aplicativo Android tem um ícone que fica na tela inicial do Android. Quando o usuário clica nesse ícone, a activity inicial do projeto é executada, que neste caso é a MainActivity.

```

<activity android:name=".MainActivity" android:label="@string/app_name">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

```

A tag <intent-filter> é necessária para customizar a forma como a activity será iniciada. A ação MAIN significa que a activity pode ser iniciada isoladamente, como o ponto inicial da aplicação. A ação MAIN é como a famosa função `main()` da linguagem C, a qual é utilizada para iniciar a aplicação. A categoria LAUNCHER indica que a activity estará disponível para o usuário na tela inicial junto com as outras aplicações que o usuário possui instaladas.

Lembre-se de que na maioria das vezes somente uma activity será configurada como o ponto de partida, portanto as outras não devem conter esse tipo de configuração. Por exemplo, para declarar uma activity com a classe `BenVindoActivity` basta uma declaração simples assim:

```
<activity android:name=".BenVindoActivity" />
```

À medida que novos exemplos forem sendo criados nos próximos capítulos, mais detalhes sobre a configuração do arquivo *AndroidManifest.xml* serão fornecidos.

Nota: sempre que criar uma nova activity é obrigatório configurá-la no arquivo *AndroidManifest.xml*. A `MainActivity` foi configurada como a activity inicial da aplicação, pois está declarada com a ação `MAIN` e a categoria `LAUNCHER`. Mais detalhes sobre a tag `<intent-filter>` serão explicados no capítulo 20, sobre a classe `Intent`.

3.3 Classe `MainActivity`

A classe `MainActivity` foi configurada pelo wizard do Android Studio como a activity inicial, portanto pode ser executada pelo usuário ao clicar no ícone do aplicativo na tela Home do Android.

A classe `MainActivity` ou qualquer outra activity deve ser filha da classe `android.app.Activity`.

```
public class MainActivity extends android.app.Activity {
    . . .
}
```

A classe `android.app.Activity` representa uma tela da aplicação, sendo responsável por controlar o estado e os eventos da tela. Assim, para cada tela da aplicação você criará uma classe-filha de `android.app.Activity`. O método `onCreate(bundle)` precisa ser implementado obrigatoriamente e é chamado de forma automática pelo Android quando a tela é criada. No entanto, a classe `android.app.Activity` não sabe desenhar nada na tela e para isso precisa da ajuda da classe `android.view.View`, que por sua vez se encarrega de desenhar os componentes visuais, como campos de texto, botões e imagens. Para isso existem diversas subclasses especializadas de `android.view.View`, conforme podemos verificar no diagrama de classes resumido mostrado na figura 3.3. No diagrama podemos verificar as classes `TextView`, `Button`, `EditText`, `ImageView` etc., responsáveis por desenhar os elementos gráficos da tela.

O método `onOptionsItemSelected(Menu menu)`, que também foi criado automaticamente na classe `MainActivity`, apenas demonstra uma simples implementação de como criar um item de menu na action bar. Já o método `onOptionsItemSelected(MenuItem item)` é chamado quando algum botão ou item de menu é clicado na action bar. Mas por enquanto não se preocupe com essa parte, pois vamos estudar tudo isso depois.

3.4 Arquivo de layout `activity_main.xml`

No Android é possível criar o layout da tela em arquivos XML ou utilizar a API Java, mas o recomendado é criar o layout em XML, para separar a lógica de negócios da camada de apresentação. Podemos dizer que a `activity` é o Controller do padrão MVC (Model View Controller) e a `view` é o arquivo XML com o layout.

Uma `view` pode ser um simples componente gráfico (botão, checkbox, imagem) ou uma `view` complexa, que atua como um gerenciador de layout, que pode conter várias `views`-filhas e tem a função de organizá-las na tela. Os gerenciadores de layout serão explicados no capítulo 6. No código a seguir, podemos visualizar o código-fonte do arquivo `/res/layout/activity_main.xml` que define a interface gráfica da tela.

```
 /res/layout/activity_main.xml

<?xml version="2.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent" // Largura da tela
    android:layout_height="match_parent" // Altura da tela
    android:paddingLeft="@dimen/activity_horizontal_margin" // Espaçamento na esquerda
    android:paddingRight="@dimen/activity_horizontal_margin" // Espaçamento na direita
    android:paddingTop="@dimen/activity_vertical_margin" // Espaçamento no topo
    android:paddingBottom="@dimen/activity_vertical_margin" // Espaçamento em baixo
    tools:context=".MainActivity">
    <TextView // Componente (view) que mostra um texto (label)
        android:text="@string/hello_world" // Mensagem
        android:layout_width="wrap_content" // Largura da view
        android:layout_height="wrap_content" /> // Altura da view
</RelativeLayout>
```

No código foi utilizada a sintaxe `@dimen` para acessar valores de espaçamento que foram definidos no arquivo `/res/values/dimens.xml`. Deixar essas constantes em arquivos XML é uma boa prática porque podemos alterar o código em um lugar centralizado, além de poder customizar o valor dessas constantes quando necessário; por exemplo, para utilizar um valor diferente para a versão `tablet` do mesmo aplicativo.

`/res/values/dimens.xml`

```
<resources>
  <!-- Espaçamento recomendado pelas boas práticas do Android Design. -->
  <dimen name="activity_horizontal_margin">16dp</dimen>
  <dimen name="activity_vertical_margin">16dp</dimen>
</resources>
```

Nota: o valor 16dp é recomendado pelas boas práticas de interface (guidelines) como o espaçamento padrão para as margens de um layout. A notação dp (density independent pixels) é uma unidade de medida do Android para tornar transparente ao desenvolvedor a quantidade de pixels utilizada em cada aparelho. No Android nunca se deve utilizar a unidade de pixels (px), pois isso pode trazer resultados diferentes dependendo da resolução da tela. Uma dica interessante é utilizar o Ctrl+Clique no arquivo XML, pois ao passar o mouse em cima das notações @dimen/valor ou @string/texto podemos abrir rapidamente o arquivo em que os recursos foram definidos.

No código-fonte do arquivo `/res/layout/activity_main.xml` podemos ver a tag `<TextView>` que mostra um simples texto na tela. Essa tag define o atributo `android:text="@string/hello_world"`, que utiliza uma mensagem identificada pela chave `hello_world` localizada no arquivo `/res/values/strings.xml`. Observe que no arquivo existem as tags `<RelativeLayout>` e `<TextView>`, que correspondem às classes `RelativeLayout` e `TextView`, respectivamente. Ambas as classes são filhas da classe `android.view.View` do Android. A classe `RelativeLayout` é um gerenciador de layout que tem o papel de organizar a disposição dos componentes. Nesse exemplo a tag `<TextView>` está contida dentro da tag `<RelativeLayout>` justamente para compor a interface da tela.

Nota: para cada tag definida no arquivo XML de layout existe uma classe correspondente que é filha de `android.view.View`. Para exemplificar, as tags `<TextView>`, `<ImageView>` e `<Button>` correspondem às classes `android.widget.TextView`, `android.widget.ImageView` e `android.widget.Button`.

Se você abrir o arquivo `/res/layout/activity_main.xml` no editor, verá que na parte inferior existem duas abas para alternar entre o código-fonte XML e o editor visual (Figura 3.4).

A figura 3.5 demonstra a opção **Preview All Screen Sizes**, que mostra no editor a pré-visualização para diferentes tamanhos de telas. Esse recurso é incrível e ajuda muito no desenvolvimento.

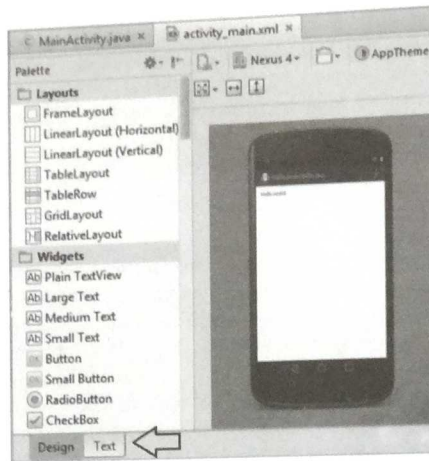


Figura 3.4 – Editor visual.

No Android você deve criar layouts que se ajustem automaticamente ao tamanho e à resolução da tela; portanto, se você desenvolver corretamente, não terá problemas com a diversidade de aparelhos. Apenas no caso dos tablets que têm um grande espaço disponível na tela talvez seja necessário um layout específico para oferecer uma melhor experiência ao usuário. Isso é feito para deixar o aplicativo mais agradável a fim de que seja utilizado no tablet, mas por padrão o layout de um aplicativo que funciona no smartphone deve funcionar perfeitamente no tablet ou até mesmo no Google TV.

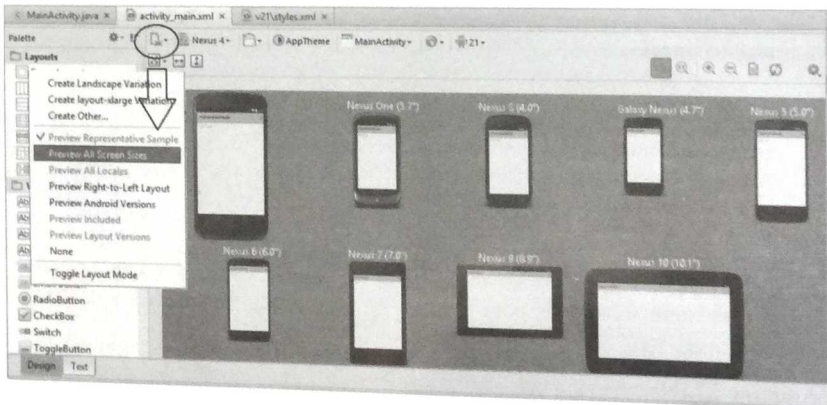


Figura 3.5 – Pre-visualização para vários dispositivos.

3.5 Arquivo strings.xml

O arquivo `/res/values/strings.xml` contém as mensagens para organizar os textos em um único arquivo centralizado, o que é uma boa prática de programação. As mensagens desse arquivo podem ser traduzidas para diversos idiomas para internacionalizar o aplicativo. Por padrão, esse arquivo contém o nome do aplicativo que digitamos ao criar o projeto e a mensagem que aparece na tela inicial.

 `/res/values/strings.xml`

```
<?xml version="2.0" encoding="utf-8"?>
<resources>
    <string name="app_name">HelloAndroidStudio</string> <!-- Nome do aplicativo -->
    <string name="hello_world">Hello world!</string> <!-- Mensagem mostrada na primeira tela -->
    <string name="action_settings">Settings</string> <!-- Label do item de menu da action bar -->
</resources>
```

O nome do aplicativo é definido pela chave `app_name`, e o texto que aparece na tela é definido pela chave `hello_world`. No arquivo `/res/layout/activity_main.xml`, ao definir a tag `<TextView>`, foi utilizado o atributo `android:text="@string/hello_world"` para referenciar a mensagem definida pela chave `hello_world`. O padrão para acessar essas mensagens, como você já deve ter percebido, é `@string/nomeDaChave`.

Para criar um aplicativo com suporte a diversos idiomas, basta criar uma pasta `/res/values/values-(<código do idioma>)` e traduzir o arquivo `/res/values/strings.xml`. A seguir temos um exemplo de uma pasta com as mensagens em inglês e outra em português.

- `/res/values-en/strings.xml`
- `/res/values-pt/strings.xml`

3.6 Classe R

A classe `R` é gerada automaticamente ao compilar o projeto e contém as constantes para acessar os diversos recursos do projeto. Sempre que um recurso é adicionado no projeto, como por exemplo uma nova imagem, a classe `R` é gerada automaticamente para conter uma constante para o novo recurso criado. No Android Studio a classe `R` é gerada na pasta `app/build/generated/source/r`.

 `R.java`

```
public final class R {
    public static final class attr { }
```

```

public static final class drawable {
    public static final int icon=0x7f020000;
}
public static final class layout {
    public static final int main=0x7f030000;
}
public static final class string {
    public static final int app_name=0x7f040001;
    public static final int hello=0x7f040000;
}
}

```

Nota: a classe R nunca deve ser alterada manualmente!

3.7 Informações sobre como acessar recursos de texto e imagem

Vamos aproveitar que falamos da classe R e explicar um pouco mais sobre a sintaxe para acessar os recursos do projeto. Para explicar alguns conceitos, veja este trecho de código do arquivo *AndroidManifest.xml*.

```

<application android:icon="@mipmap/ic_launcher" // Ícone do aplicativo na tela Home
    android:label="@string/app_name" ... > // Nome da aplicação na tela Home

```

A tag `android:icon` define o ícone do aplicativo que vai ficar na tela inicial do Android, sendo que a sintaxe `@drawable` é utilizada para acessar a imagem */res/drawable/ic_launcher.png*. Essa sintaxe é padrão para acessar imagens sempre que você estiver em algum arquivo XML. Por exemplo, ao adicionar uma view do tipo `ImageView` no arquivo */res/layout/activity_main.xml*, podemos mostrar uma figura utilizando a notação `@drawable/nome_figura`:

```

<ImageView android:src="@drawable/nome_da_figura"
    android:layout_width="wrap_content" android:layout_height="wrap_content" />

```

Porém, caso você precise acessar esse `ImageView` e atualizar a imagem dinamicamente pelo código Java, vai precisar utilizar a classe R. Primeiro precisamos definir um identificador para o `ImageView` com a tag `android:id` e a sintaxe `@+id/identificador`.

```

<ImageView android:id="@+id/img"
    android:layout_width="wrap_content" android:layout_height="wrap_content" />

```

Com este `id` é possível utilizar o método `findViewById(id)` no código da *activity* para acessar esse objeto, e depois disso é utilizada a classe R para acessar o recurso da imagem com a constante `R.drawable.nome_imagem`.

```
// Encontra o objeto pelo id
ImageView img = (ImageView) findViewById(R.id.img);
// Atualiza a imagem dinamicamente. A classe R é utilizada para acessar o recurso.
img.setImageResource(R.drawable.nome_da_imagem);
```

Agora vamos falar um pouco de mensagens e o arquivo */res/values/strings.xml*. Se você reparar novamente no código do arquivo *AndroidManifest.xml*, verá que o nome da aplicação foi definido pela chave `@string/app_name`. A mesma sintaxe pode ser utilizada ao criar o layout dos arquivos em XML. Por exemplo, no arquivo */res/layout/activity_main.xml* foi criado um `TextView` que referencia a mensagem `@string/hello_world`, conforme demonstrado a seguir.

```
<TextView android:text="@string/hello_world"
    android:layout_width="wrap_content" android:layout_height="wrap_content" />
```

Da mesma forma que fizemos com o `ImageView`, também é possível atualizar o texto de um `TextView` dinamicamente. Para acessar o objeto do `TextView` no código, novamente é preciso definir um identificador para a view com a tag `android:id`.

```
<TextView android:id="@+id/text"
    android:layout_width="wrap_content" android:layout_height="wrap_content" />
```

Para acessar esse objeto `TextView` no código, basta utilizar o método `findViewById(id)`. Note que a constante `R.string.hello_world` é utilizada com objetivo de acessar a mensagem `hello_world` que está definida no arquivo */res/values/strings.xml*.

```
// Encontra o objeto pelo id
TextView text = (TextView) findViewById(R.id.text);
text.setText(R.string.hello_world); // Atualiza o texto dinamicamente.
```

A fim de encerrar o assunto, vamos fazer uma breve revisão. Lembre-se de que a notação com o caractere `@` é utilizada sempre que for necessário acessar um recurso em um arquivo XML. Caso o recurso precise ser acessado no código Java, é preciso utilizar a classe `R`. A tabela 3.1 compara as duas formas de acessar os recursos:

Tabela 3.1 – Formas de acessar um recurso

Objetivo	Utilizando XML	Utilizando a classe R
Acessar a imagem <i>foto.png</i> localizada na pasta <i>/res/drawable</i>	<code>@drawable/foto</code>	<code>R.drawable.foto</code>
Acessar a mensagem <i>hello</i> definida no arquivo <i>/res/values/strings.xml</i>	<code>@string/hello</code>	<code>R.string.hello</code>

3.8 Arquivo build.gradle

O sistema de build do Android é baseado no Gradle. No projeto existe o arquivo *build.gradle* padrão de todos os módulos e o arquivo *app/build.gradle* com as configurações de compilação do módulo *app*, que é onde fica o código-fonte do aplicativo.

A seguir, podemos visualizar o arquivo *build.gradle* que fica na raiz do projeto. Na prática, você não vai alterar esse arquivo, exceto se o plugin do Gradle for atualizado, pois no arquivo fica o código da versão do plugin.

build.gradle

```
buildscript {
    repositories {
        jcenter() // Repositório padrão do Android
    }
    dependencies {
        // Versão do plugin do Gradle. Se o plugin for atualizado atualize este código
        // Caso tenha um erro de compilação nessa linha, tecle Alt+Enter para abrir o assistente
        // O assistente vai ajudá-lo a configurar a versão correta do plugin
        classpath 'com.android.tools.build:gradle:1.0.0'
    }
}
allprojects {
    repositories {
        jcenter() // Repositório para todos os projetos
    }
}
```

O arquivo mais importante é o *app/build.gradle* que fica dentro da pasta do módulo *app* do projeto, lugar em que fica o código-fonte da aplicação. No arquivo *app/build.gradle* você configura a versão do aplicativo e também a versão mínima do Android (API Level) que seu aplicativo suporta, além de declarar as bibliotecas que são necessárias para a compilação. A seguir podemos visualizar o código-fonte do arquivo *app/build.gradle*.

app/build.gradle

```
apply plugin: 'com.android.application' // Aplica o plugin 'android' no script de
                                        // compilação do Gradle
android { // Este elemento configura os parâmetros de compilação do módulo
    compileSdkVersion 22 // API Level usada para compilar o código (deve ser sempre a última)
    buildToolsVersion "22.0.1" // Versão do Android SDK Build-tools baixada pelo SDK Manager
}
```

```
defaultConfig { // Este elemento configura os itens do AndroidManifest.xml
    applicationId "br.com.livroandroid.helloandroidstudio" // Pacote do projeto
                                                    // (identificador único da aplicação)
    minSdkVersion 22 // API Level mínima que o aplicativo suporta.
    targetSdkVersion 22 // API Level máxima utilizada para otimizar o build na compilação
    versionCode 1 // Código para identificar o aplicativo no Google Play
    versionName "1.0" // Versão do código com formato amigável para o usuário
}
buildTypes { // Este elemento configura os tipos de build (debug e release)
    release { // Configura o build do tipo release
        minifyEnabled false // Configura para utilizar o proguard e obfuscar o código
    }
}
}
dependencies { // Declara as dependências (bibliotecas) para compilar do projeto
    // Inclui todos os arquivos .jar da pasta libs na compilação
    compile fileTree(dir: 'libs', include: ['*.jar'])
}
```

No final do arquivo *app/build.gradle* são declaradas as dependências do projeto, mas isso vamos exercitar bastante durante o livro, portanto fique tranquilo. A parte mais importante desse arquivo de configuração são todos os itens que ficam dentro do elemento *defaultConfig*. Nesse elemento são configurados: o pacote do aplicativo, a versão mínima do SDK e o código de versão do aplicativo. Antigamente essas configurações ficavam no arquivo *AndroidManifest.xml*, mas agora tudo é gerenciado pelo Gradle.

Com o Gradle é possível criar builds customizados do aplicativo, para, por exemplo, criar versões que apontem para web services de homologação ou produção. Todo esse poder e flexibilidade para fazer o build é uma das vantagens do Gradle, além de gerenciar as dependências, é claro.

Nota: todo aplicativo (apk) deve ser assinado com um certificado digital antes de ser instalado no dispositivo. Por padrão o sistema de build do Gradle compila o projeto no modo *debug* e *release*. O modo *debug* é utilizado durante o desenvolvimento, e o aplicativo é assinado com o certificado digital de desenvolvimento, o qual é criado automaticamente pelo Android Studio e fica na pasta do usuário *~/.android/debug.keystore*. O modo *release* é utilizado para publicar o aplicativo no Google Play e deve ser assinado com outro certificado, o qual você precisa criar. Vamos estudar mais detalhes sobre isso no capítulo 38, sobre Gradle.

3.9 LogCat – Escrevendo mensagens de log

Em Java, para imprimir uma mensagem no console é utilizado o comando `System.out.println(string)`, mas no Android é recomendado utilizar a classe `android.util.Log`, que contém métodos utilitários para imprimir informações com os níveis de detalhes desejados, como informação (i), debug (d), warning (w) e erro (e).

Dica: no Android Studio, utilize a tecla de atalho `Alt+6` para abrir a janela `6:Android`. Nessa janela você verá os logs do LogCat, que tem por finalidade gerenciar todos os logs do sistema operacional.

Com a classe `android.util.Log` é possível criar logs de informação, debug, alertas e erro. Cada log pode ser escrito em uma determinada tag, para que posteriormente apenas as mensagens desejadas sejam recuperadas. Para demonstrar como criar logs em diferentes níveis de severidade, altere o código-fonte da classe `MainActivity` conforme demonstrado a seguir.



MainActivity.java

```
...
import android.util.Log;

public class MainActivity extends Activity {
    private static final String TAG = "livro";
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main_layout);
        ...
        Log.v(TAG, "log de verbose");
        Log.d(TAG, "log de debug");
        Log.i(TAG, "log de info");
        Log.w(TAG, "log de alerta");
        Log.e(TAG, "log de erro", new RuntimeException("teste de erro"));
    }
}
```

Observe que a constante `TAG` definida na classe é utilizada como identificador das mensagens. Isso é útil para filtrar as mensagens por tags/categorias, facilitando a visualização e debug. Os métodos da classe `Log` são bem simples, e começam com a primeira letra do nível de severidade. Por exemplo, para imprimir um log com nível de alerta é utilizado o método `Log.w(tag, mensagem)`. O `w` é da palavra `warning`

(alerta). É importante entender que é necessário informar a tag do log, para que depois seja possível filtrar os logs apenas da tag desejada.

Agora execute o código e visualize os logs gerados na janela LogCat (Figura 3.6). Observe que mensagens vermelhas no LogCat são erros. Neste exemplo lancei uma exceção no código para você visualizar um erro de teste, portanto acostume-se a olhar o LogCat, pois qualquer exceção lançada pelo seu aplicativo será detalhada aqui.

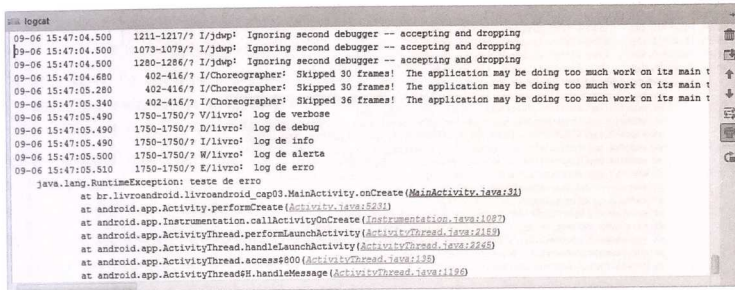


Figura 3.6 – Janela LogCat.

Contudo, podemos verificar que nos logs não aparecem apenas as mensagens que criamos, pois também aparecem todas as outras mensagens do sistema operacional. Mas foi exatamente por isso que criamos os logs utilizando a tag `livro`, pois agora podemos filtrar as mensagens apenas dessa tag.

Para criar um filtro, procure no canto direito superior da janela LogCat o combo com os filtros e clique em `Edit Filter Configuration`. Feito isso, crie uma configuração de filtro para a tag `livro`, conforme mostra a figura 3.7.

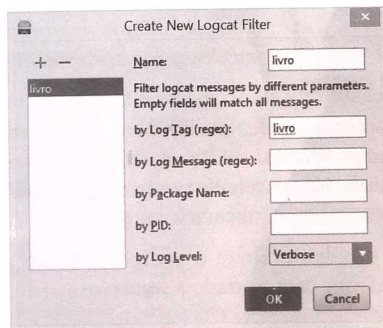


Figura 3.7 – Criação de um filtro para o LogCat.

Nesse caso o nome da tag é livro, pois é o valor da constante TAG que definimos na classe.

```
private static final String TAG = "livro";
```

Depois de criar o filtro, é possível visualizar somente os logs gerados pelo nosso aplicativo (Figura 3.8). Agora podemos ver somente as mensagens que nos interessam.

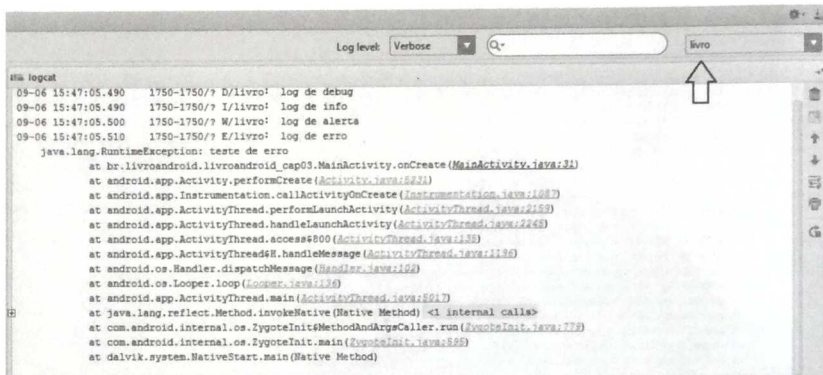


Figura 3.8 – Janela do LogCat exibindo as mensagens do filtro desejado.

Importante: acostume-se a utilizar o LogCat, pois é nessa janela que todas as mensagens de logs e erros são exibidas. Lembre-se de que, em caso de erro, o “stack trace” das exceções aparece no LogCat, e você pode inclusive verificar o número da linha que originou o problema. Sempre que seu aplicativo travar (force close), olhe as mensagens de erro no LogCat. Lembre-se de não utilizar o `System.out.println` diretamente, pois é recomendado utilizar o LogCat e separar os logs em nível de severidade e tags.

3.10 Tratamento de eventos

Neste próximo exemplo vamos alterar o layout da `MainActivity` para criar uma tela de login com dois campos para o usuário e senha, a fim de fazer a validação desse login de forma simulada. Neste exemplo vamos aprender não só a tratar o evento de clique em um botão, como também a ler os valores digitados pelo usuário.

Para começar a brincadeira, altere o código-fonte do arquivo `/res/layout/activity_main.xml` conforme demonstrado a seguir:

```

 /res/layout/activity_main.xml

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="wrap_content"
    android:gravity="center_vertical" android:orientation="vertical"
    android:padding="16dp">
    <TextView
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="@string/usuario" />
    <!-- Campo de texto para digitar o login -->
    <EditText android:id="@+id/tLogin"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:inputType="text" android:singleLine="true" />
    <TextView
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:layout_marginTop="10dp" android:text="@string/senha" />
    <EditText android:id="@+id/tSenha"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:inputType="textPassword" android:singleLine="true" />
    <Button android:id="@+id/btLogin"
        android:layout_width="200dp" android:layout_height="wrap_content"
        android:layout_marginTop="6dp" android:text="@string/login"
        android:textColor="#ffffff" android:layout_gravity="center"/>
</LinearLayout>

```

A tag raiz desse layout é um `LinearLayout`, que é um gerenciador de layout que organiza as views na horizontal ou vertical. Nesse caso a orientação foi configurada como vertical com o atributo `android:orientation="vertical"`, portanto as views serão posicionadas uma abaixo da outra. A tag `<TextView>` é um simples label e a tag `<EditText>` é um campo de entrada de dados para o usuário digitar um valor. A tag `<Button>`, acredito que dispensa apresentações.

O importante é você perceber que no layout foi adicionado um id para os dois campos de texto e também para o botão. Isso é feito com a tag `android:id`. No código vamos utilizar o método `findViewById(id)` para acessar esses objetos pelo id. A seguir podemos verificar o código-fonte atualizado da classe `MainActivity` que trata o evento no botão e faz o login para o usuário “ricardo” e senha “123”.

 `MainActivity.java`

```

...
// Faça os imports das classes necessárias aqui.
// Caso não esteja familiarizado com Java, utilize o assistente do Android Studio.

```

```
// Na linha do erro você pode clicar neste assistente para fazer o import.
// O atalho ALT+Enter abre o assistente se o cursor estiver sobre a linha.
// O menu Code > Optimize Imports também pode ajudar.
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button btLogin = (Button) findViewById(R.id.btLogin);
        btLogin.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                TextView tLogin = (TextView) findViewById(R.id.tLogin);
                TextView tSenha = (TextView) findViewById(R.id.tSenha);
                String login = tLogin.getText().toString();
                String senha = tSenha.getText().toString();
                if("ricardo".equals(login) && "123".equals(senha)) {
                    alert("Bem-vindo, login realizado com sucesso.");
                } else {
                    alert("Login e senha incorretos.");
                }
            }
        });
    }
    private void alert(String s) {
        // A classe Toast mostra um alerta temporário muito comum no Android
        Toast.makeText(this,s,Toast.LENGTH_SHORT).show();
    }
}
```

No Android, para tratar os eventos de um botão é utilizado o método `setOnClickListener(listener)`. Esse método recebe como argumento uma instância da interface `android.view.View.OnClickListener`. A interface `View.OnClickListener` define o método `onClick(view)`, o qual é chamado quando o evento ocorrer, passando como argumento o objeto da view que gerou o evento, neste caso, o botão.

Nota: para definir um id para os componentes é utilizada a notação `android:id="@+id/identificador_aqui"`. O método `findViewById(id)` é utilizado no código para encontrar uma view pelo seu id.

Depois dessa alteração, execute o projeto para conferir o resultado (Figura 3.9). Ao digitar o usuário "ricardo" e senha "123", o login de teste é realizado.

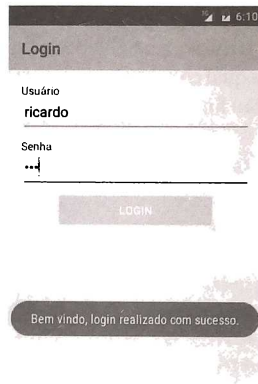


Figura 3.9 – Tratamento de eventos.

Desenvolvedores com alguma experiência devem ter percebido que o tratamento de eventos no Android é similar a qualquer outra linguagem. Na verdade, existem muitas formas de escrever esse mesmo código, basta informar ao método `setOnClickListener(listener)` algum objeto que implementa a interface `android.view.View.OnClickListener`.

O código anterior utilizou o conceito de classes anônimas do Java para fazer essa implementação. Outra maneira de tratar o evento é fazer a classe da activity implementar a interface `View.OnClickListener`, e utilizar o `this (self)` para referenciar a própria instância da classe para tratar o evento.



MainActivity.java

```
...
public class MainActivity extends Activity implements View.OnClickListener {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button btLogin = (Button) findViewById(R.id.btLogin);
        button.setOnClickListener(this); // O this representa a instância da classe
    }
    @Override
    public void onClick(View view) { // A própria classe implementa View.OnClickListener
        // Código que trata o evento aqui
    }
}
```

A desvantagem dessa implementação é que, se você tiver mais de um botão na tela, o mesmo método `onClick(view)` será chamado para todos os botões. Nesse caso o parâmetro `View` indica qual componente gerou o evento, e deve ser utilizado para descobrir em qual botão foi feito o clique, conforme demonstrado a seguir.



MainActivity.java

```
...
public class MainActivity extends Activity implements OnClickListener {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main_layout);
        Button btOk1 = (Button) findViewById(R.id.botaoOk);
        Button btOk2 = (Button) findViewById(R.id.botaoOk2);
        // Vamos imaginar que existem dois botões na tela
        btOk1.setOnClickListener(this);
        btOk2.setOnClickListener(this);
    }
    @Override
    public void onClick(View view) {
        if(view.getId() == R.id.botaoOk) {
            // clicou no botão 1
        } else if(view.getId() == R.id.botaoOk2) {
            // clicou no botão 2
        }
    }
}
```

Vejamos outra forma de escrever o mesmo código, que é criar um método para cada botão. Eu particularmente gosto desse jeito. A vantagem é que fica simples de separar cada clique em um método diferente.



MainActivity.java

```
...
public class MainActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main_layout);
        Button btLogin = (Button) findViewById(R.id.btLogin);
        btLogin.setOnClickListener(onClickLogin());
    }
}
```

```
// O método onClickLogin() retorna uma implementação de View.OnClickListener
private View.OnClickListener onClickLogin() {
    return new Button.OnClickListener() {
        public void onClick(View v) {
            // Tratar o evento de clique aqui.
        }
    };
}
}
```

Dica: acostume-se a utilizar o assistente de código com a tecla de atalho **Alt+Enter** para complementar o texto do código. Nesse exemplo eu digitei o método `onClickLogin()` e todo o restante foi criado com o assistente.

Ainda existe uma última maneira de tratar os eventos. Existem desenvolvedores que preferem definir o nome do método que deve ser chamado no arquivo XML de layout.

```
<Button android:id="@+id/btLogin"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_margin="6dp" android:text="Login"
        android:onClick="onClickBtLogin" />
```

Nesse caso o atributo `android:onClick="onClickBtLogin"` informa que o método `onClickBtLogin(view)` será chamado na classe da activity e deve existir, conforme demonstrado a seguir:

```
public void onClickBtLogin(View view) {
    // Trata o evento de clique aqui
}
```

Para gerar o código do método na activity, novamente o assistente do Android Studio pode ajudá-lo. No arquivo de layout, clique na linha em que o atributo `android:onClick="onClickBtLogin"` foi definido e tecle **Alt+Enter**. Você verá uma opção para criar o método na activity.

Eu não recomendo fazer desta última maneira, pois se o nome do método for digitado de modo incorreto você descobrirá o erro somente ao executar o projeto. Em meus projetos, prefiro definir os métodos no código, pois qualquer problema é descoberto em tempo de compilação. Em muitos exemplos deste livro, acabei usando essa técnica de definir o método no XML, pois ela reduz a quantidade de código a fim de facilitar a leitura.



CAPÍTULO 4

Activity

A classe `Activity` representa uma tela da aplicação e é responsável por controlar os eventos da tela e definir qual `View` será responsável por desenhar a interface gráfica do usuário.

Neste capítulo, vamos estudar como navegar entre telas da aplicação, como é feita a passagem de parâmetros de uma tela para outra e o ciclo de vida da classe `Activity`.

4.1 Activity

Uma `activity` é uma classe que deve herdar da classe `android.app.Activity` ou de alguma subclasse desta, a qual representa uma tela da aplicação e é responsável por tratar os eventos gerados nessa tela.

A classe da `activity` deve sobrescrever o método `onCreate(bundle)`. Esse método é obrigatório e responsável por realizar a inicialização necessária para executar a aplicação, como por exemplo chamar o método `setContentView(view)` para definir a interface de usuário.

Cada `activity` deve ser obrigatoriamente declarada no arquivo *AndroidManifest.xml*. Isso é feito por meio da tag `<activity>`, como demonstrado a seguir.

```
<activity android:name=".MinhaClasseActivity" />
```

Não há muito o que falar da classe `android.app.Activity`. Apenas que é utilizada para construir uma tela da aplicação. Assim, se você está pensando em criar uma nova tela, vai precisar de uma nova `activity`. A coisa mais importante que você precisa lembrar é que, sempre que uma nova classe de `activity` for criada, ela precisa ser declarada no arquivo *AndroidManifest.xml*.

Nessa configuração, a `activity` geralmente é declarada com a sintaxe do ponto, o que indica que o pacote da classe é relativo ao pacote do projeto. Por exemplo, se

o pacote do projeto é `br.com.livroandroid.cap4` e a classe também está nesse pacote, podemos utilizar a sintaxe do ponto.

```
<activity android:name=".MinhaClasseActivity" />
```

No entanto, se a classe da activity estiver em outro pacote, como por exemplo `br.com.livroandroid.cap4.activity`, podemos declarar essa activity com a seguinte sintaxe:

```
<activity android:name=".activity.MinhaClasseActivity" />
```

Se achar necessário, definir o nome completo da classe também não é errado:

```
<activity android:name="br.livro.android.cap4.activity.MinhaClasseActivity" />
```

Nota: não erre o nome da classe da activity na configuração do *AndroidManifest.xml* e lembre-se de que o nome precisa ser exatamente igual, pois é case-sensitive. Uma boa dica para ter certeza de que o nome está correto é segurar a tecla Ctrl e passar o mouse em cima do nome. Se o texto ficar selecionado, você pode clicar para abrir o código-fonte da classe no editor. Se o editor abrir tudo está correto.

Na prática podemos associar a palavra “activity” à palavra “tela”. Por exemplo, vamos analisar as seguintes frases, na qual a primeira está escrita com a palavra “activity” e a segunda com a palavra “tela”.

1. Iniciar uma activity: “iniciar uma tela”.
2. O sistema operacional decidiu encerrar a activity X para economizar memória: “O sistema operacional decidiu encerrar a tela X para economizar memória”.

Seguindo esse raciocínio, se for necessário criar uma nova tela no projeto, você saberá que é preciso criar uma nova activity. A tradução da palavra activity para o português é atividade. Então, também podemos dizer que uma activity representa uma atividade, ação ou funcionalidade que o usuário pode realizar dentro de sua aplicação. Se preferir, sempre que ler a palavra “activity” em uma frase pode interpretá-la como “atividade”. Prefiro manter a palavra em inglês neste livro para maior consistência.

4.2 Classes `FragmentActivity` e `AppCompatActivity`

Existem algumas subclasses famosas da classe `android.app.Activity` que é bom você conhecer desde já. Agora talvez o conceito não fique claro, pois estamos só começando, mas futuramente você pode voltar neste tópico e lê-lo novamente.

Estou explicando isso porque o Android Studio pode gerar um código diferente, dependendo da versão do Android que você selecionar no wizard de criação de projetos.

Então vamos lá! No Android 3.0 foram criadas a action bar e a API de fragments.

A action bar é representada pela classe `android.app.ActionBar` e um fragment pela classe `android.app.Fragment`. Para utilizar essas classes em versões mais antigas do Android o Google criou a biblioteca de compatibilidade. A biblioteca de compatibilidade v4 (API Level 4) é compatível com o Android 1.6 ou superior, e dentre as APIs que ela tem está contida a API de fragments. Mas para usar os fragments de compatibilidade todas as activities do projeto precisam herdar de `android.support.v4.app.FragmentActivity`, que é uma subclasse especial da classe `android.app.Activity` que permite utilizar os fragments nas versões antigas do Android. Ao utilizar a biblioteca v4 você deve utilizar a classe `android.support.v4.app.Fragment` para acessar os fragments e não a classe nativa `android.app.Fragment`.

Da mesma forma, existe a biblioteca de compatibilidade v7 (API Level 7), que é compatível com o Android 2.3 ou superior. A biblioteca v7 contém a action bar de compatibilidade e permite habilitar a action bar nos dispositivos mais antigos. Para isso todas as activities do projeto precisam herdar de `android.support.v7.app.AppCompatActivity`. Ao utilizar a biblioteca v7 você deve utilizar a classe `android.support.v7.app.ActionBar` para acessar a action bar e não a classe nativa `android.app.ActionBar`.

Então basicamente temos três classes de activity que você precisa conhecer.

- `android.app.Activity` – Classe padrão da activity.
- `android.support.v4.app.FragmentActivity` – Classe da biblioteca de compatibilidade v4, que permite utilizar os fragments em versões antigas do Android.
- `android.support.v7.app.AppCompatActivity` – Classe da biblioteca de compatibilidade v7, que permite utilizar a action bar em versões antigas do Android. A classe `AppCompatActivity` é filha de `FragmentActivity`.

Como exercício, você pode criar outro projeto no Android Studio e escolher a API Level 9 (Android 2.3) como a mínima do projeto. Ao fazer isso, o Android Studio vai criar o projeto utilizando a biblioteca de compatibilidade v7, e nesse caso a `MainActivity` será filha de `AppCompatActivity`. Bem, teoricamente era isso que deveria acontecer, porém a versão do Android Studio que utilizo ao escrever este livro gerava o código antigo.

```
public class MainActivity extends android.support.v7.app.ActionBarActivity {
```

O correto seria usar a classe `AppCompatActivity`:

```
public class MainActivity extends android.support.v7.app.AppCompatActivity {
```

O wizard gera a activity de compatibilidade corretamente, porém ele usa a `ActionBarActivity` em vez da `AppCompatActivity`. O problema é que em 21 de abril de 2015 o Google postou no blog que a `ActionBarActivity` foi descontinuada (deprecated) a favor da `AppCompatActivity`. É possível que quando você estiver lendo este livro o wizard do Android Studio já esteja gerando o código atualizado, pois isso aconteceu quando eu estava terminando de revisar o livro.

Nota: não se preocupe com essas questões de compatibilidade agora, estou apenas introduzindo o assunto para você ter uma ideia. Por enquanto utilize a classe `android.app.Activity` nativa nos exemplos. Durante seus estudos, ao criar um novo projeto sempre utilize a maior versão do Android disponível, pois vou informá-lo quando precisar criar um projeto com compatibilidade.

As dependências para essas bibliotecas são declaradas no arquivo *build.gradle*, mas vamos deixar para mostrar esses exemplos mais tarde, pois ainda precisamos estudar o básico sobre activity.

Nos capítulos sobre Action Bar (5) e Fragments (8) vamos aprender a utilizar as bibliotecas de compatibilidade, portanto fique tranquilo. Por ora, lembre-se de que no capítulo 2 eu pedi para você criar o projeto no Android Studio sempre utilizando a última versão, assim estamos utilizando a classe padrão da activity, que é a `android.app.Activity`.

Pelo menos agora você já conhece essas três classes do tipo activity, e quando encontrar algum código que as utilize terá uma ideia do porquê. Esse tópico foi apenas para alertá-lo disso, mas, como eu já disse, fique tranquilo porque vamos estudar tudo isso depois.

4.3 Ciclo de vida de uma activity

Para ser um bom desenvolvedor Android, é muito importante entender o ciclo de vida de uma activity, isto é, os possíveis estados que ela assume. O importante é entender que o sistema operacional cuida desse ciclo de vida, mas ao desenvolver aplicações é importante levar cada estado possível em consideração para desenvolver uma aplicação mais robusta.

Por exemplo, digamos que você criou um maravilhoso jogo no Android e, enquanto o usuário está jogando, alguém faz uma ligação para ele. Nesse caso, o usuário vai parar o jogo para atender a ligação, não é? E o que acontece com o jogo? O sistema operacional vai parar o jogo temporariamente (pause) e colocá-lo em

segundo plano enquanto o usuário atende a ligação. Depois que a ligação terminar, o sistema operacional reiniciará a aplicação do jogo. Agora, a grande pergunta é: será que seu jogo vai conseguir continuar de onde parou? Será que o estado e informações do jogo serão salvos ou tudo será perdido? O Android fornece toda a estrutura necessária para controlar este estado, basta você entender o ciclo de vida de uma activity e implementar os métodos corretamente.

Uma activity tem um ciclo de vida bem definido. Cada activity iniciada é inserida no topo de uma pilha, chamada de “activity stack” ou, se preferir traduzir, “pilha de atividades”. O conceito de pilha já é bem conhecido por todos. Assim, sempre que uma nova activity é inserida no topo da pilha, a activity anterior que estava em execução fica logo abaixo da nova. A activity que está no topo da pilha é a activity que está em execução no momento, as demais podem estar executando em segundo plano, estar no estado pausado ou totalmente paradas.

Sobre essa pilha é importante você entender que no Android até a tela inicial (Home) é uma activity. Na verdade, sempre que você abre um aplicativo, a activity que representa aquela tela é inserida no topo da pilha. Então digamos que você esteja na tela inicial do Android e clicou no ícone do Browser. Depois você clica no ícone do Gmail. Nesse momento temos a seguinte pilha: **Home > Browser > Gmail**, sendo que a aplicação da tela inicial é sempre a base da pilha. Nesse caso alguma activity da aplicação do Gmail está executando no topo da pilha, e a activity do Browser está parada em segundo plano (pause).

Sempre que uma activity está pausada o sistema operacional pode decidir encerrar o processo, para, por exemplo, liberar recursos e memória para outras aplicações. Por exemplo, digamos que o usuário estava jogando e de repente decidiu navegar na internet e, para isso, ele parou o jogo. Isso faz com que o Android insira no topo da pilha a aplicação do browser, e deixe em segundo plano o jogo que está temporariamente parado. Agora, digamos que o usuário esqueça-se de voltar ao jogo para continuá-lo, e, por algum motivo, o sistema operacional precisa liberar recursos e memória. Sendo assim, como o jogo não está mais em uso, o sistema operacional pode decidir encerrar seu processo. Nesse momento sua aplicação pode decidir salvar algumas informações ou não.

Nota: cada aplicação nativa do Android, como o browser, a tela inicial (Home), a agenda e a própria tela para discar para um número de telefone é definida por uma Activity. Portanto, esse conceito de pilha de atividades é utilizado para todas as aplicações, sejam as desenvolvidas por você ou uma aplicação nativa, pois tudo funciona sobre a mesma arquitetura.

Agora partiremos para a prática. Existem métodos da classe Activity que podem ser utilizados para controlar o estado da aplicação. Dentre eles já vimos o método `onCreate(bundle)` que é responsável por inicializar a activity e dar início ao seu ciclo de vida. Agora vamos estudar outros métodos importantes: `onCreate(bundle)`, `onStart()`, `onRestart()`, `onResume()`, `onPause()`, `onStop()` e `onDestroy()`. A figura 4.1 demonstra o ciclo de vida completo de uma activity, exibindo os possíveis estados e a chamada de cada método.

Fonte:

<http://developer.android.com/training/basics/activity-lifecycle/starting.html>

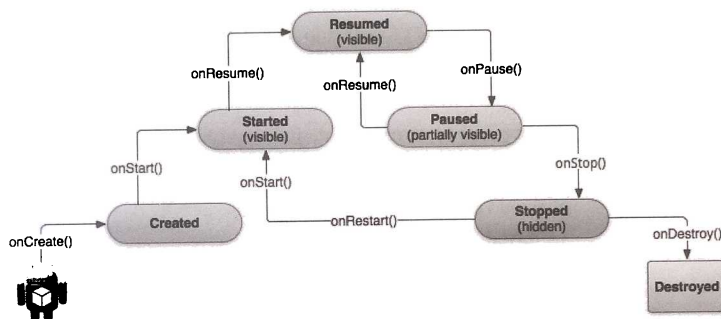


Figura 4.1 – Ciclo de vida de uma activity.

Depois de ver o diagrama da figura 4.1, leia atentamente o significado de cada método do ciclo de vida da activity.

Método	Descrição
<code>onCreate(bundle)</code>	O método <code>onCreate(bundle)</code> é obrigatório e chamado uma única vez. O objetivo desse método é fazer a inicialização necessária para executar a aplicação. Nele deve-se criar uma view e chamar o método <code>setContentView(view)</code> para configurar o layout da tela.
<code>onStart()</code>	O método <code>onStart()</code> é chamado quando a activity está ficando visível ao usuário e já tem uma view. Pode ser chamado depois dos métodos <code>onCreate()</code> ou <code>onRestart()</code> , dependendo do estado da aplicação.
<code>onRestart()</code>	O método <code>onRestart()</code> é chamado quando uma activity foi parada temporariamente e está sendo iniciada outra vez.
<code>onResume()</code>	O método <code>onResume()</code> é chamado quando a activity está no topo da pilha “activity stack”, e dessa forma já está executando como a activity principal e interagindo com o usuário. Podemos dizer que o método <code>onResume()</code> representa o estado de que a activity está executando. É importante você entender que quando o método <code>onResume()</code> executar

Método	Descrição (cont.)
	o usuário já está vendo a tela. Por isso esse método é frequentemente utilizado para disparar threads que consultam os dados em web services ou banco de dados para atualizar as informações da tela.
<code>onPause()</code>	Sempre que a tela da activity fechar, o método <code>onPause()</code> será chamado. Isso pode acontecer se o usuário pressionar o botão Home ou o botão voltar para sair da aplicação. Ou também pode acontecer se você receber uma ligação telefônica. Nesse momento o método <code>onPause()</code> é chamado para salvar o estado da aplicação, para que posteriormente, quando a activity voltar a executar, tudo possa ser recuperado, se necessário, no método <code>onResume()</code> .
<code>onStop()</code>	O método <code>onStop()</code> é chamado logo depois do método <code>onPause()</code> e indica que a activity está sendo encerrada, e não está mais visível ao usuário. Depois de parada, a activity pode ser reiniciada se necessário. Caso isso ocorra, o método <code>onRestart()</code> é chamado para reiniciar a activity. Caso a activity fique muito tempo parada em segundo plano e o sistema operacional do Android precise limpar os recursos para liberar memória, o método <code>onDestroy()</code> pode ser automaticamente chamado para remover completamente a activity da pilha.
<code>onDestroy()</code>	O método <code>onDestroy()</code> literalmente encerra a execução de uma activity. O método <code>onDestroy()</code> pode ser chamado automaticamente pelo sistema operacional para liberar recursos ou pode ser chamado pela aplicação pelo método <code>finish()</code> da classe <code>Activity</code> . Depois de o método <code>onDestroy()</code> ter executado, a activity é removida completamente da pilha e seu processo no sistema operacional também é completamente encerrado.

Para demonstrar as chamadas dos métodos do ciclo de vida da activity, vamos criar a classe `DebugActivity`, que sobrescreve cada método e insere um log do `LogCat`. Para este próximo exemplo vamos criar o projeto `HelloActivity`, mas se preferir abra o projeto pronto dos exemplos do livro. O código-fonte da classe `DebugActivity` pode ser visualizado a seguir:



`DebugActivity.java`

```

...
// Activity que imprime logs nos métodos de ciclo de vida
public class DebugActivity extends Activity {
    protected static final String TAG = "livro";
    protected void onCreate(Bundle icle) {
        super.onCreate(icle);
        Log.i(TAG, getClass().getName() + ".onCreate() chamado: " + icle);
    }
}

```

```

protected void onStart() {
    super.onStart();
    Log.i(TAG, getClass().getName() + ".onStart() chamado.");
}
protected void onRestart() {
    super.onRestart();
    Log.i(TAG, getClass().getName() + ".onRestart() chamado.");
}
protected void onResume() {
    super.onResume();
    Log.i(TAG, getClass().getName() + ".onResume() chamado.");
}
protected void onPause() {
    super.onPause();
    Log.i(TAG, getClass().getName() + ".onPause() chamado.");
}
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    Log.i(TAG, getClass().getName() + ".onSaveInstanceState() chamado.");
}
protected void onStop() {
    super.onStop();
    Log.i(TAG, getClass().getName() + ".onStop() chamado.");
}
protected void onDestroy() {
    super.onDestroy();
    Log.i(TAG, getClass().getName() + ".onDestroy() chamado.");
}
private String getClass().getName() {
    // Retorna o nome da classe sem o pacote
    String s = getClass().getName();
    return s.substring(s.lastIndexOf("."));
}
}

```

Nota: sempre que sobrescrever um método da classe Activity, chame o método da classe mãe com o super, caso contrário uma exceção será lançada.

Essa classe imprime um log quando cada método do ciclo de vida for chamado. O log é criado com a tag livro, portanto é necessário criar um filtro para essa tag na janela do LogCat, conforme explicado no capítulo 3. O próximo passo é alterar

a classe `MainActivity` para ser filha de `DebugActivity`, assim ela vai herdar todos os métodos que foram customizados na sua classe mãe.



MainActivity.java

```
...
public class MainActivity extends DebugActivity {
    // Mesmo código aqui
}
```

Ao executar o projeto no emulador, não estamos interessados no layout da activity, então desta vez vamos analisar somente os logs que mostram as chamadas para cada método do ciclo de vida. Na primeira vez que a aplicação executar, os seguintes logs são gerados. Observe que os métodos `onCreate()`, `onStart()` e `onResume()` são chamados na sequência.

```
INFO/ID(8494): MainActivity.onCreate() chamado.
INFO/ID(8494): MainActivity.onStart() chamado.
INFO/ID(8494): MainActivity.onResume() chamado.
```

Agora clique no botão voltar (back) do emulador para sair da activity. Isso pode ser feito também pressionando a tecla **Esc**. Observe que agora os métodos `onPause()`, `onStop()` e `onDestroy()` foram chamados para encerrar o ciclo de vida da activity.

```
INFO/ID(8494): MainActivity.onPause() chamado.
INFO/ID(8494): MainActivity.onStop() chamado.
INFO/ID(8494): MainActivity.onDestroy() chamado.
```

Nesse caso, como o botão voltar foi pressionado, o sistema operacional sabe que a activity precisa ser destruída. Por isso, o método `onDestroy()` foi chamado, eliminando completamente a activity da memória.

Agora volte na tela inicial do emulador e entre novamente na aplicação. Isso vai chamar os métodos `onCreate()`, `onStart()` e `onResume()`. Então, clique no botão **Home** para voltar à tela inicial. Observe que nos logs os métodos `onPause()` e `onStop()` foram chamados, mas não o método `onDestroy()`.

```
INFO/ID(8670): MainActivity.onPause() chamado.
INFO/ID(8670): MainActivity.onStop() chamado.
```

Isso acontece porque a activity não foi completamente destruída, apenas retirada do topo da pilha, e agora está parada em segundo plano.

Por último, volte à tela inicial Home e clique no ícone da aplicação novamente. Isso vai reiniciar a activity, fazendo com que ela volte ao topo da pilha para ser executada em primeiro plano. Assim, os métodos `onRestart()`, `onStart()` e `onResume()` são chamados.

```
INFO/ID(10550): MainActivity.onRestart() chamado.  
INFO/ID(10550): MainActivity.onStart() chamado.  
INFO/ID(10550): MainActivity.onResume() chamado.
```

Observe que o método `onCreate(bundle)` é chamado uma única vez. Se a activity estiver parada em segundo plano, o método `onCreate(bundle)` não é chamado novamente. Já o método `onResume()` é chamado sempre que a tela ficar visível ao usuário.

Perceba que os métodos `onPause()` e `onStop()` são sempre chamados ao sair da tela. O método `onResume()` é chamado sempre que iniciar ou voltar para a aplicação.

O método `onCreate()` é chamado apenas na primeira vez que a aplicação é criada. Caso a activity seja totalmente destruída, evento que acontece ao clicar no botão voltar, o método `onDestroy()` é chamado para eliminar os recursos e encerrar o ciclo de vida da activity. Outra forma de destruir a activity é chamar o método `finish()` programaticamente no código.

Entender o funcionamento do ciclo de vida de uma activity é muito importante para desenvolver aplicações no Android. Temos de estar preparados para ações externas que podem ocorrer, tal como uma ligação telefônica para o usuário. Essa ação faz com que o sistema operacional do Android interrompa (pause/stop) a activity atual, colocando-a em segundo plano. Isso é feito porque a aplicação nativa da ligação é quem vai ocupar o topo da pilha de atividades. Numa situação como essa, quando a ligação terminar, temos de voltar e executar a aplicação de onde ela parou sem perder nenhuma informação.

Nota: entender o ciclo de vida de uma activity é fundamental para dominar o desenvolvimento para Android. É isso que diferencia um bom desenvolvedor de um desenvolvedor comum.

4.4 Ciclo de vida avançado – o que acontece ao rotacionar o celular?

Ao girar a tela do celular da vertical para a horizontal é importante você saber que o Android vai destruir a activity atual e vai recriá-la logo em seguida. O Android faz isso porque ele precisa recriar todas as views e aplicar espaçamentos e margens adequadas para a nova orientação.

Durante esse processo de troca de orientação, o Android vai chamar o método `onSaveInstanceState(bundle)` na classe da activity. Esse método recebe um objeto do tipo `android.os.Bundle` como argumento que deve ser utilizado para armazenar os

dados em uma estrutura de chave e valor. Para demonstrar esse comportamento, vamos executar a aplicação novamente. Ao fazer isso, a `MainActivity` é criada e inserida no topo da pilha, conforme mostram estes logs:

```
MainActivity.onCreate() chamado: null
MainActivity.onStart() chamado.
MainActivity.onResume() chamado.
```

Agora pressione a tecla de atalho **Ctrl+F11** do emulador para girar a tela para a horizontal. Os logs a seguir mostram que a activity foi destruída e recriada, mas durante esse processo o método `onSaveInstanceState(bundle)` foi chamado.

```
MainActivity.onPause() chamado.
MainActivity.onSaveInstanceState() chamado.      // Salve o estado aqui
MainActivity.onStop() chamado.
MainActivity.onDestroy() chamado.                // Activity foi destruída
MainActivity.onCreate() chamado: Bundle[{}]]] // Activity foi recriada. Recupere o
                                                // estado aqui

MainActivity.onStart() chamado.
MainActivity.onResume() chamado.                  // Activity está visível para o usuário
```

Se você salvou valores no `Bundle` lá no método `onSaveInstanceState(bundle)` é possível recuperar este estado no `Bundle` que você recebe como parâmetro no método `onCreate(bundle)`. É para isso que serve esse `Bundle` no método `onCreate(bundle)`. Se for a primeira vez que a activity é executada, esse `Bundle` sempre estará nulo. Entretanto, no caso da rotação da tela em que o Android faz esse processo para recriar a activity, o `Bundle` pode estar preenchido com os dados salvos no método `onSaveInstanceState(bundle)`.

Nota: o comportamento padrão do Android ao girar a tela é destruir a activity atual e recriar outra. É sua responsabilidade manter o estado da aplicação salvando os dados no `Bundle` durante a execução do método `onSaveInstanceState(bundle)`, para depois recuperar o estado no método `onCreate(bundle)`.

Essa teoria inicial é para você já ficar esperto, mas no capítulo 8, sobre fragments, vamos revisar esse conceito com exercícios práticos.

4.5 Navegação entre telas e inicialização de uma nova activity

Geralmente um aplicativo é composto de várias telas, então precisamos aprender como fazer a navegação entre as telas.

Existem dois métodos que podem ser utilizados para iniciar outra activity/tela: `startActivity(intent)` e `startActivityForResult(intent,código)`. A diferença entre eles é que o método `startActivity(intent)` apenas inicia a próxima activity sem qualquer vínculo. O método `startActivityForResult(intent,código)` recebe um parâmetro que identifica essa chamada, para que posteriormente essa segunda activity possa retornar alguma informação para a activity que a chamou. Esse método é utilizado caso a activity inicial que fez a chamada esteja interessada em obter o retorno quando a segunda activity terminar.

Esse retorno de uma activity para outra pode ser utilizado, por exemplo, em uma aplicação que exibe uma tela para escolher um contato do celular e, logo depois de escolher o contato, a informação é retornada para a activity inicial, para, por exemplo, enviar uma mensagem ou email para o contato selecionado.

Os métodos `startActivity(intent)` e `startActivityForResult(código,resultado,intent)` recebem um objeto do tipo `android.content.Intent` como parâmetro. A classe `android.content.Intent` é o coração do Android, tudo gira em torno dela, e uma explicação detalhada será fornecida no capítulo 20.

Então mãos à obra! Crie uma nova classe chamada `BemVindoActivity`, que também deve herdar de `DebugActivity`, assim podemos monitorar as chamadas dos métodos do ciclo de vida. Para criar uma activity é preciso fazer três coisas:

1. Criar a classe Java que estende de alguma subclasse de `android.app.Activity`.
2. Criar o arquivo XML de layout.
3. Registrar a activity no arquivo `AndroidManifest.xml`.

Para facilitar esse trabalho, recomendo você clicar com o botão direito no pacote das classes do projeto e selecionar o wizard **New > Activity > Blank Activity** (Figura 4.2). Esse wizard vai auxiliá-lo a executar estes três passos.

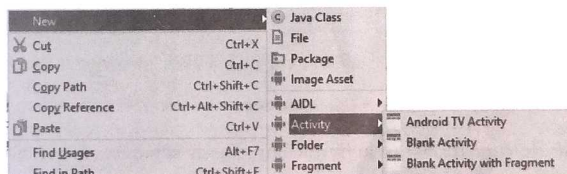


Figura 4.2 – Wizard para criar uma activity.

Depois de criar a activity com o wizard, digite o seguinte código-fonte:



BemVindoActivity.java

```
import android.app.Activity;
import android.os.Bundle;

public class BemVindoActivity extends DebugActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_bem_vindo); // Layout desta activity
        // Recebe o nome enviado por parâmetro
        Bundle args = getIntent().getExtras();
        String nome = args.getString("nome");
        // Vamos atualizar o texto do TextView com uma mensagem de bem-vindo
        TextView text = (TextView) findViewById(R.id.text);
        text.setText(nome + ", seja bem-vindo.");
    }
}
```

No layout da activity vamos adicionar um simples `TextView` com o id `android:id="@+id/text"`, para atualizar o texto dinamicamente dentro do código.



/res/layout/activity_bem_vindo.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" android:padding="10dp" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
</LinearLayout>
```

Nota: o arquivo de layout XML segue a nomenclatura `/res/layout/activity_nome_classe.xml`.

Lembre-se de que todas as activities precisam ser declaradas no arquivo *AndroidManifest.xml*. A vantagem de utilizar o wizard do Android Studio é que ele já fez essa configuração.



AndroidManifest.xml

```

...
<application ...>
    <activity android:name=".MainActivity" ... >
        ... // intent-filter da MAIN aqui.
    </activity>
    <activity android:name=".BemVindoActivity" android:label="Bem-vindo" />
</application>

```

Nota: todas as activities precisam ser declaradas no arquivo *AndroidManifest.xml*. Geralmente somente uma activity terá as configurações de `<intent-filter>` MAIN e LAUNCHER, pois é a activity que fica na tela inicial e é chamada pelo usuário.

O próximo passo é alterar o código da classe *MainActivity* para navegar para a activity *BemVindoActivity*. Na classe *MainActivity* fizemos o exemplo do layout de login e vamos continuar de onde paramos. Caso o login seja feito com sucesso, vamos navegar para a próxima tela, a qual vai mostrar uma mensagem de bem-vindo ao usuário. Nesse exemplo vou passar um parâmetro `nome=Ricardo` fixo para a próxima tela.



MainActivity.java

```

...
public class MainActivity extends DebugActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button btLogin = (Button) findViewById(R.id.btLogin);
        btLogin.setOnClickListener(onClickLogin());
    }
    private View.OnClickListener onClickLogin() {
        return new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                TextView tLogin = (TextView) findViewById(R.id.tLogin);
                TextView tSenha = (TextView) findViewById(R.id.tSenha);
                String login = tLogin.getText().toString();
                String senha = tSenha.getText().toString();
                if("ricardo".equals(login) && "123".equals(senha)) {
                    // Navega para a próxima tela
                    Intent intent = new Intent( getContext(), BemVindoActivity.class);

```

```

        Bundle params = new Bundle();
        params.putString("nome", "Ricardo Lecheta");
        intent.putExtras(params);
        startActivity(intent);
    } else {
        alert("Login e senha incorretos.");
    }
}
};
}
private Context getContext() {
    return this;
}
private void alert(String s) {
    Toast.makeText(this,s,Toast.LENGTH_SHORT).show();
}
}

```

Para navegar para a próxima tela, é criado um objeto do tipo `android.content.Intent` informando a classe da activity que deve ser chamada. Ao criar a intent, é preciso passar a referência do contexto, que é a classe `android.content.Context`, que por sua vez é mãe de `android.content.Activity`.

Por isso, geralmente é comum ver no código-fonte o contexto sendo referenciado com o `this` da classe, pois o `this` no Java representa a instância do objeto atual. No caso de o código estar dentro de uma classe Activity do Android, o `this` representa essa activity.

```

Intent it = new Intent(this, BemVindoActivity.class);
startActivity(it);

```

Eu muitas vezes gosto de declarar a variável context conforme demonstrado a seguir. Neste exemplo simples não tem muita utilidade, mas em exemplos mais complexos vale a pena.

```

final Context context = this;
Intent it = new Intent(context, BemVindoActivity.class);
startActivity(it);

```

Veja que no código do método `onClick(view)` da `MainActivity` não foi utilizado o `this` para passar o contexto. Isso porque o método `onClick(view)` criou uma classe interna (inner class do Java), e nesse caso o `this` faz referência à classe interna e não à classe `MainActivity`. Por isso criei o método `getContext()` para retornar o `this` e facilitar o código.

Ao executar a aplicação e fazer o login, é feita a navegação para a tela de bem-vindo, conforme a figura 4.3. Como você já deve saber, para voltar à tela anterior basta utilizar o botão de voltar.

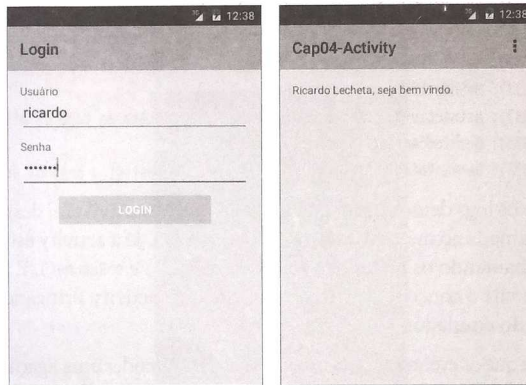


Figura 4.3 – Navegação de telas com a activity.

Agora vamos voltar a falar do ciclo de vida da activity. Ao abrir a segunda activity, ela é inserida no topo da pilha de atividades e a tela do login está parada em segundo plano. Para confirmar a teoria, vamos verificar os logs na janela LogCat.

Ao abrir a primeira activity, podemos verificar nos logs que os métodos `onCreate()`, `onStart()` e `onResume()` foram chamados normalmente como esperado.

```
INFO/ID(11033): MainActivity.onCreate() chamado.
INFO/ID(11033): MainActivity.onStart() chamado.
INFO/ID(11033): MainActivity.onResume() chamado.
```

Até agora não há novidades. Mas ao fazer o login os seguintes logs aparecem no LogCat:

```
INFO/ID(11033): MainActivity.onPause() chamado. // Primeira activity é interrompida (pause)
INFO/ID(11033): BemVindoActivity.onCreate() chamado // Segunda activity é iniciada
INFO/ID(11033): BemVindoActivity.onStart() chamado.
INFO/ID(11033): BemVindoActivity.onResume() chamado. // Segunda activity é colocada no
// topo da pilha
INFO/ID(11033): MainActivity.onStop() chamado. // Primeira activity é parada (stop)
```

Podemos verificar que a activity `MainActivity` teve seus métodos `onPause()` e `onStop()` chamados para deixar a primeira tela em segundo plano. Já a activity `BemVindoActivity` teve os seus métodos de inicialização chamados e agora ocupa o topo da pilha.

Entretanto, o que acontece se o botão voltar for pressionado? O sistema operacional identificará que a activity `BemVindoActivity` deve ser destruída, porque teoricamente ela não é mais necessária. Isso fará com que a activity `MainActivity` seja reiniciada, voltando ao topo da pilha. Esse fluxo pode ser verificado pelos seguintes logs:

```
INFO/ID(11033): BemVindoActivity.onPause() chamado. // A segunda activity é parada
INFO/ID(11033): MainActivity.onRestart() chamado. // Reinicia a primeira activity
INFO/ID(11033): MainActivity.onStart() chamado.
INFO/ID(11033): MainActivity.onResume() chamado. // Volta ao topo da pilha
INFO/ID(11033): BemVindoActivity.onStop() chamado.
INFO/ID(11033): BemVindoActivity.onDestroy() chamado. // Destroi a segunda activity
```

Observe que os logs demonstram que a activity `BemVindoActivity` foi destruída, uma vez que a chamada ao método `onDestroy()` foi realizada. Já a activity `MainActivity` foi reiniciada, chamando os métodos `onRestart()`, `onStart()` e `onResume()`. É importante que você exercite o conceito de ciclo de vida de uma activity, brincando bastante com os logs do emulador.

Lembre-se de que os eventos externos, como o usuário atender uma ligação telefônica, também podem parar a activity e deixá-la executando em segundo plano. Os métodos do ciclo de vida devem estar preparados caso isso aconteça. Já que o exemplo da ligação foi lembrado, vamos aproveitar e finalizar o assunto com chave de ouro e simular uma chamada telefônica, para que a activity que está executando seja inserida em segundo plano enquanto o usuário atende a ligação. Caberá a você executar a activity, analisar os logs e estudar o que ocorre quando o usuário atende uma chamada.

Para simular a chamada telefônica, abra a ferramenta **Android Device Monitor** no menu **Tools > Android > Android Device Monitor** e procure pela janela **Emulator Control**. Nessa janela digite um número de telefone no campo **Incoming Number** e clique no botão **Call** (Figura 4.4).

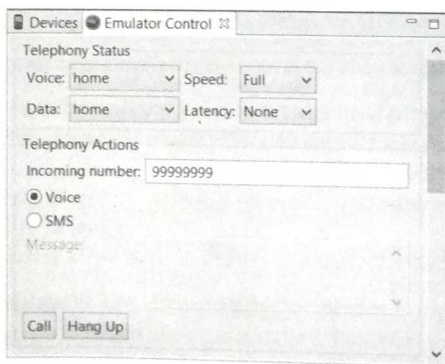


Figura 4.4 – Simulando uma ligação.

A figura 4.5 mostra o emulador simulando uma ligação telefônica. A foto do Mickey apareceu nessa figura porque no capítulo 32, sobre a agenda de contatos (Content Provider), eu cadastrei os contatos Mickey, Donald e Pateta no meu emulador.

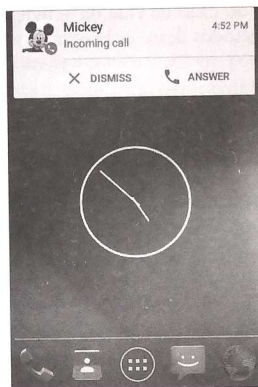


Figura 4.5 – Emulador recebendo uma ligação simulada.

Lembre-se: se alguma activity estiver em execução enquanto a ligação for feita, ela será removida do topo da pilha e ficará parada em segundo plano. Agora quem ocupa o topo da pilha é a própria aplicação nativa da ligação. Como assim, a própria aplicação da ligação? Isso mesmo, a tela que exibe a ligação, assim como a tela inicial, o browser, os mapas, a agenda de contatos etc., todas elas são uma activity, só que nativas e já disponíveis no Android. Aos poucos você vai entendendo que no Android tudo funciona da mesma forma, seja uma aplicação desenvolvida por você, seja uma nativa do Android.

Voltando à ligação, quando ela terminar e o usuário fechar a tela, a activity que estava parada em segundo plano será reiniciada e voltará ao topo da pilha, estando pronta para interagir com o usuário novamente. Esse comportamento é incrível, já que o sistema operacional encarrega-se de tudo. Ele coloca a activity em segundo plano e depois a reinicia quando necessário, e, é claro, todos os métodos de ciclo de vida são chamados para o seu controle.

Depois de toda essa teoria não se esqueça de brincar um pouco com o emulador: é muito importante entender o funcionamento do ciclo de vida da classe Activity.

Nota: se existirem muitos processos parados em segundo plano e se as condições de memória estiverem baixas, o sistema operacional pode decidir encerrar o processo de uma activity, chamando o método `onDestroy()`. O fato é que o sistema operacional encarrega-se do ciclo de vida da activity. Cabe ao desenvolvedor apenas implementar os métodos desse ciclo corretamente.

4.6 Mais detalhes sobre a classe `Bundle` e como passar parâmetros

No exemplo anterior, aprendemos a utilizar o método `startActivity(intent)` para navegar entre as telas e vimos que a classe `android.os.Bundle` é utilizada para passar parâmetros usando a estrutura de chave e valor.

Para revisar o código que fizemos, veja como foi criado um `Bundle` e nele foi passado o parâmetro `nome=Ricardo`. Depois é chamado o método `intent.putExtras(bundle)` para passar esses parâmetros para a `intent`:

```
...
public class MainActivity extends DebugActivity {
    ...
    if("ricardo".equals(login) && "123".equals(senha)) {
        Intent intent = new Intent(getApplicationContext(), BemVindoActivity.class);
        Bundle params = new Bundle();
        params.putString("nome", "Ricardo Lecheta");
        intent.putExtras(params);
        startActivity(intent);
    }
    ...
}
```

Nesse exemplo uma string foi adicionada como parâmetro, mas a classe `Bundle` tem métodos para diferentes tipos primitivos, como por exemplo: `putBoolean`, `putChar`, `putByteArray`, `putShort`, `putInt`, `putLong`, `putFloat`, `putDouble` e vários outros.

Se quiser você pode escrever esse código de maneira mais simplificada, pois a classe `Intent` tem alguns atalhos para passar parâmetros. Nesse caso podemos substituir o trecho de código anterior por apenas três linhas. Internamente a classe `Intent` vai continuar utilizando o `Bundle`, mas isso fica encapsulado.

```
public class MainActivity extends DebugActivity {
    ...
    if("ricardo".equals(login) && "123".equals(senha)) {
        Intent intent = new Intent(getApplicationContext(), BemVindoActivity.class);
```

```

        intent.putExtra("nome", "Ricardo Lecheta");
        startActivity(intent);
    }
    ...
}

```

Para ler o parâmetro na segunda activity, também podemos utilizar um atalho, e ler diretamente da classe `Intent`, conforme demonstrado a seguir:

```

Intent intent = getIntent();
String nome = intent.getStringExtra("nome"); // Lê o parâmetro do Bundle

```

Ou você pode recuperar o objeto `Bundle` completo para ler os parâmetros.

```

Intent intent = getIntent();
Bundle args = intent.getExtras();
String nome = args.getString("nome");

```

Existem vários métodos para cada tipo primitivo, como `getIntExtra(chave)`, `getLongExtra(chave)`, `getBooleanExtra(chave)` etc. Agora que você já conhece a classe `Bundle` e os métodos da classe `Intent` que encapsulam o acesso, fica a seu critério utilizar a forma de enviar e recuperar os parâmetros que mais lhe agradar.

4.7 O básico sobre action bar e como voltar para tela anterior

No último exemplo, mostramos como navegar entre duas activities da aplicação. Sempre que uma activity é chamada, surge a necessidade de voltar à tela anterior, e para isso você pode utilizar o botão de voltar do Android, ou colocar o indicador de voltar na action bar.

Para adicionar o indicador de voltar na action bar, basta utilizar o método `getActionBar().setDisplayHomeAsUpEnabled(true)`, e quando o usuário clicar no botão ele vai disparar a ação de menu com o identificador `android.R.id.home`. Para testar a brincadeira, altere o código da classe `BemVindoActivity` conforme demonstrado a seguir:

```

BemVindoActivity.java

...
public class BemVindoActivity extends DebugActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_bem_vindo);
        Bundle args = getIntent().getExtras();
        String nome = args.getString("nome");
    }
}

```

```

        TextView text = (TextView) findViewById(R.id.text);
        text.setText(nome + ", seja bem-vindo.");
        // Adiciona o botão "up navigation"
        getActionBar().setDisplayHomeAsUpEnabled(true);
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();
        if(id == android.R.id.home) {
            // O método finish() vai encerrar essa activity
            finish();
            return true;
        }
        return super.onOptionsItemSelected(item);
    }
}

```

Nota: o método `getActionBar()` retorna o objeto `android.app.ActionBar` que está disponível para o Android 3.0 (API Level 11) ou superior. Se estivéssemos utilizando a biblioteca de compatibilidade, seria utilizado o método `getSupportActionBar()`. Posteriormente vamos aprender a manter a compatibilidade com versões antigas do Android, mas por enquanto não se preocupe com isso.

Desta vez, ao executar o projeto você verá que na action bar da segunda activity apareceu a seta que aponta para esquerda (Figura 4.6), indicando que o usuário pode voltar para a tela anterior. O indicador de voltar na action bar é chamado de “up navigation”, e vai adicionar uma pequena seta para esquerda indicando que o usuário pode clicar nela para subir na hierarquia de telas.

O evento gerado pelo up navigation é como se o usuário tivesse clicado em qualquer outro botão da action bar. Nesse caso, porém, o identificador do evento é a constante `android.R.id.home` da classe `android.R` nativa do Android. Observe que no código, como temos de voltar manualmente para a primeira activity, é chamado o método `finish()`, que faz com que a activity atual seja encerrada. Assim, o Android vai remover a activity da pilha e chamar todos os métodos do ciclo de vida como `onPause()`, `onStop()`, até o `onDestroy()`.

Antes de continuarmos o assunto, deixe-me explicar algo importante sobre o up navigation. Segundo as boas práticas de design do Android, existe uma diferença entre o botão voltar do Android e o up navigation. O botão voltar sempre volta para a tela anterior e pronto. Mas o up navigation conceitualmente faz a aplicação subir na hierarquia de telas. Por exemplo, se o celular está parado na tela Home do

Android e sua aplicação for chamada por meio de uma notificação (como se chegasse um email), você pode decidir como o up navigation deve funcionar. O botão voltar simplesmente vai voltar para a tela anterior (que nesse exemplo é a home), mas o up navigation pode voltar para uma tela diferente (tela com a lista de emails).

Muitas vezes você também pode ter várias telas empilhadas: A > B > C > D > E. Na tela E, se você pressionar o botão voltar, as telas serão desempilhadas uma a uma até a tela A. Mas utilizando o up navigation podemos mudar esse comportamento e fazer a aplicação voltar direto para a tela A. Isso fica a critério da sua aplicação; o importante é você entender que o up navigation nem sempre funcionará igual ao simples botão voltar do Android.

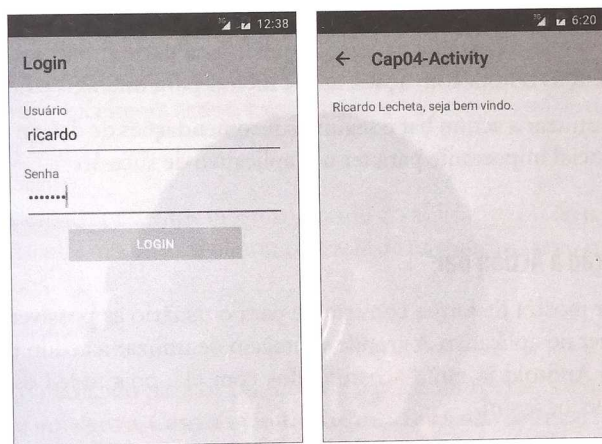


Figura 4.6 – Botão de up navigation na action bar.

Nota: acostume-se com o termo up navigation, o qual se refere à seta de voltar que é colocada na parte superior esquerda da action bar. Outro detalhe importante: para encerrar uma activity por programação utilize o método `finish()`. Ao fazer isso, o método `onDestroy()` da activity será chamado para encerrar o seu ciclo de vida.

4.8 Links úteis

Para complementar sua leitura, segue o link da documentação oficial:

- **Android API Guides – Activities**

<http://developer.android.com/guide/components/activities.html>



CAPÍTULO 5

Action Bar e temas

Este capítulo é sobre o padrão de design mais importante do Android, que é a action bar, ou seja, a barra de navegação que fica na parte superior da aplicação, a qual contém os botões com ações, tabs e menus para interagir com o usuário.

Aprender a utilizar a action bar e seguir as recomendações de design do Android é um diferencial importante para ter um aplicativo de sucesso.

5.1 Introdução à Action Bar

A action bar mostra de forma consistente para o usuário as possíveis ações que ele pode fazer no aplicativo. A grande vantagem de utilizar a action bar é que os usuários de Android já estão acostumados com ela, pois todos os aplicativos nativos são feitos assim.

Para entender a action bar, vamos estudar a figura 5.1.



Figura 5.1 – Action bar.

1. App Icon

Por padrão o ícone da action bar mostra o ícone do projeto e pode ser customizado conforme sua necessidade. Nesse espaço também é mostrado o “up navigation”, que é a seta para esquerda que indica que o usuário pode navegar para cima na hierarquia de telas.

Esse ícone atualmente me deixou um pouco intrigado, e me fez pensar um pouco antes de escrever as próximas frases. Desde que a action bar foi criada, a documentação oficial reforça o conceito de que esse ícone representa algo importante, como o logo da aplicação. Porém, com o surgimento do Material Design, a action bar nos dispositivos com Android 5.0 ou superior não mostra o ícone, diferentemente de como era no Android 3.x e 4.x. No Material Design é dado um foco maior na cor principal da aplicação (primary color) *versus* a cor de acentuação (accent color) para dar destaque às views e a alguns componentes. No Material Design é uma boa prática customizar as cores da action bar para se identificar com a marca do cliente ou aplicativo. Também é recomendado utilizar a cor de acentuação para destacar as principais ações quando necessário.

Por isso, dependendo da versão do Android, você verá aplicativos que mostram o ícone na action bar ou não.

2. View control

Nesse espaço podemos mostrar o título do aplicativo ou da tela em que o usuário está, ou utilizar algum controle de navegação como o drop-down e as tabs.

3. Action buttons

Espaço dedicado para os botões que representam as ações mais comuns do seu aplicativo. Caso a quantidade de ícones não se encaixe no espaço reservado, automaticamente eles são inseridos no menu do action overflow.

4. Action overflow

Menu flutuante que mostra as ações que não são tão frequentes no aplicativo.

5.2 Temas Holo e Material

O segredo para utilizar a action bar está no tema que o aplicativo utiliza, o qual é configurado no arquivo *AndroidManifest.xml*.

A interface do Android funciona com base nos temas. Desde sua criação os dois temas clássicos eram o `Theme.Black` e `Theme.Light`. O tema `Theme.Black` utiliza o fundo preto e fonte branca e o tema `Theme.Light` é o contrário, utiliza o fundo branco e a fonte preta.

Esses eram os temas até o Android 2.x, mas a partir do Android 3.0 (Honeycomb) foi criado o tema **Holographic**, popularmente conhecido apenas como **Holo**. O tema **Holo** também tem suas variações com fundo preto ou branco, que são os temas **Theme.Holo** e **Theme.Holo.Light**.

Já no Android 5 (Lollipop) foi criado o tema **Material**, que aplica as regras do **Material Design**. Segundo o Google esta foi a maior mudança da história referente a design na plataforma do Android. O tema **Material** também tem suas variações com fundo preto ou branco, que são os temas **Theme.Material** e **Theme.Material.Light**.

Entender o que é um tema é muito importante para você ser um bom desenvolvedor Android, por isso, tentarei explicar desde o básico, inclusive voltando um pouco na história desde as primeiras versões do Android.

A maneira mais fácil de você entender o que é um tema é abrir um arquivo de layout no editor visual. No editor é possível configurar o tipo do dispositivo para fazer a pré-visualização, configurar a orientação (vertical ou horizontal), o idioma caso você tenha preparado o aplicativo para internacionalização etc. Uma das configurações que podemos fazer no editor é escolher o tema para visualizar a tela. Por padrão, o editor utiliza o tema **AppTheme**, conforme indicado na figura 5.2.

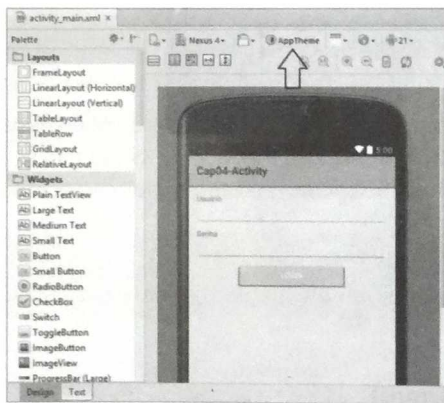


Figura 5.2 – Pré-visualização do layout com o tema padrão.

Por enquanto vamos deixar para lá este **AppTheme**, depois voltamos a falar dele. Agora vamos brincar um pouco com o editor. Para trocar o tema que o editor utiliza, clique no combo do **AppTheme** e escolha um dos temas **Classic > Black** ou **Classic Light > Light**. Esses são os temas utilizados até o Android 2.x (Figura 5.3). Observe que nesse tema não existe a action bar, pois antigamente na barra superior era mostrado apenas um pequeno título, que inclusive muitas aplicações deixavam oculto.

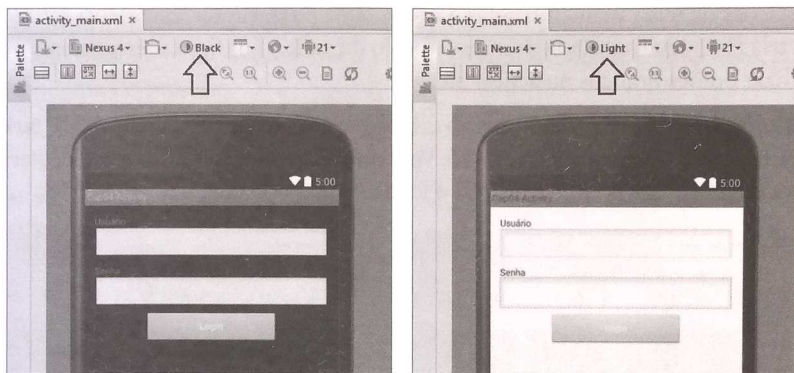


Figura 5.3 – Pré-visualização do layout com os temas do Android 2.x.

Com a chegada do Android 3.0 (API Level 11) e os temas Theme.Holo e Theme.Holo.Light a história mudou, e nasceu a action bar, que representa o padrão de design mais importante do Android, pois é nela que ficam os botões com as principais ações do aplicativo. Para fazer a pré-visualização do layout nos temas Holo e Holo.Light basta selecionar a versão do tema desejado no editor (Figura 54).

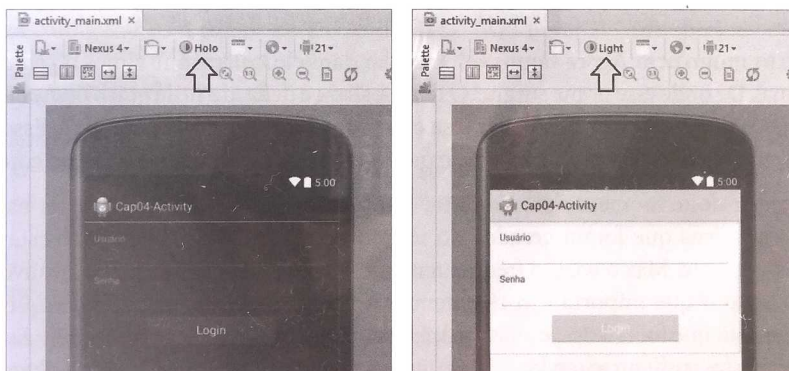


Figura 5.4 – Pré-visualização do layout com o tema Holo.

Na pré-visualização você vai perceber que o tema Holo padrão tem fundo preto e a fonte tem uma cor azul parecida com a do filme *Tron: O Legado*. Já o tema Holo.Light tem o fundo branco com fonte preta, e a action bar fica com um fundo cinza. É importante entender que no tema Holo (escuro) os botões da action bar devem ser brancos. E no tema Holo.Light (claro) os botões da action bar devem ser escuros, assim a visualização dos botões da action bar ficam coerentes. Cada tema tem diversas variações; no entanto, isso você vai perceber com o tempo. Uma das

variações de tema mais utilizadas é o `Holo.Light.DarkActionBar`, que deixa o fundo da tela claro igual o `Holo.Light`, mas o fundo da action bar fica escuro, permitindo utilizar botões brancos na action bar.

Por último você pode selecionar no editor para visualizar o tema `Material Dark` e `Material Light`, conforme a figura 5.5. Veja que a diferença entre o tema `Material` para o `Holo` é que o `Holo` mostrou o ícone na action bar.

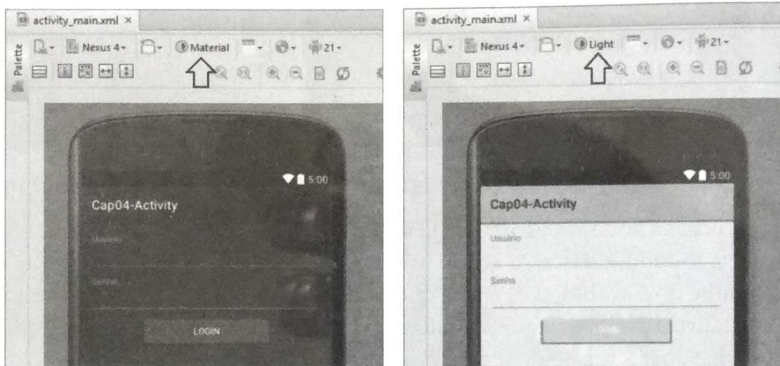


Figura 5.5 – Pré-visualização do layout com o tema `Material`.

Certo, muito bem. Agora vamos voltar a falar daquele `AppTheme` que vimos no editor visual. De onde esse nome surgiu? O tema `AppTheme` é definido no arquivo `/res/values/styles.xml`, e o editor consegue ler essa configuração e mostrá-la para você. Esse é o tema da aplicação, portanto, é nele que vamos fazer as customizações de cores.

A partir deste momento preste muita atenção, pois estou explicando com base nos arquivos que foram gerados no wizard na época em que este livro estava sendo escrito. Mas o wizard frequentemente muda a forma de gerar os arquivos, portanto o que importa é você entender o conceito. A seguir temos o arquivo `styles.xml`, que foi criado pelo wizard, é nele que o tema `AppTheme` foi definido. Esse arquivo foi configurado para utilizar o tema `Holo`. Veja que o tema `AppTheme` herda de `Theme.Holo.Light`.



`/res/values/styles.xml`

```
<resources>
    <!-- Tema mãe da aplicação. Deve herdar de algum tema conhecido. -->
    <style name="AppTheme" parent="android:Theme.Holo.Light" >
        <!-- Configure as propriedades do tema aqui. -->
    </style>
</resources>
```

E no arquivo *AndroidManifest.xml* o tema *AppTheme* foi configurado como o tema do projeto com a notação *@style/tema*. Por isso o editor visual lê esse *AppTheme*, pois ele é o tema do projeto.



AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
  <application android:theme="@style/AppTheme" >
    <!-- Activities -->
  </application>
</manifest>
```

Nota: o tema que o aplicativo vai utilizar é configurado no *AndroidManifest.xml*. Por padrão, o tema *AppTheme* é criado automaticamente no arquivo */res/values/styles.xml* e herda de algum tema nativo do Android, seja alguma variação dos temas *Holo* ou *Material*.

Veja que o Android Studio também criou o arquivo */res/values-v21/styles.xml*, que está configurado para utilizar o tema *Material*. A pasta */res/values-v21* é utilizada para configurar os recursos para o Android 5.0 (API Level 21) ou superior.



/res/values-v21/styles.xml

```
<resources>
  <!-- Tema mãe da aplicação. Deve herdar de algum tema conhecido. -->
  <style name="AppTheme" parent="android:Theme.Material.Light">
    <!-- Configure as propriedades do tema aqui. -->
  </style>
</resources>
```

No Android, sempre que você ver uma pasta com um traço e um número, como por exemplo */res/values-v21*, é porque esse número é referente ao identificador da API Level com o qual essa pasta é compatível. Nesse caso, o Android vai utilizar por padrão o arquivo */res/values/styles.xml*, e se o dispositivo tiver o Android 5.0 ou superior (API Level 21), a configuração feita no arquivo */res/values-v21/styles.xml* é utilizada.

Nota: os identificadores numéricos nas pastas são chamados de qualificadores por API Level e definem a versão do Android que vai utilizar esses recursos.

Para encerrar esta breve explicação sobre temas, lembre-se de que no arquivo *AndroidManifest.xml* o tema da aplicação é definido pelo atributo `android:theme="@style/nome_do_tema"`. Isso é feito assim para que o tema possa ser customizado dependendo da versão do Android.

Embora você possa customizar o tema dependendo da versão do Android com os qualificadores por API Level e criar variações da pasta */res/values*, atualmente isso não é mais necessário, pois podemos utilizar a biblioteca de compatibilidade e usar o tema *AppCompat* para todas as versões do Android. Vamos primeiro estudar como utilizar a action bar nativa e depois aprenderemos mais detalhes sobre essas questões de compatibilidade.

5.3 Projeto de exemplo sobre action bar

Para estudarmos o funcionamento da action bar, abra o projeto **HelloActionBar** no Android Studio, o qual faz parte do material de download do livro. Para fazer isso, utilize o menu **File Open** e selecione a pasta do projeto.

Ao executar esse projeto no emulador, o resultado será como a figura 5.6, que mostra os botões de busca e atualizar na action bar, bem como a ação de **Settings** no menu action overflow. Saiba que os botões que ficam como action buttons podem ter imagens, mas os botões que ficam no menu action overflow contêm apenas textos.

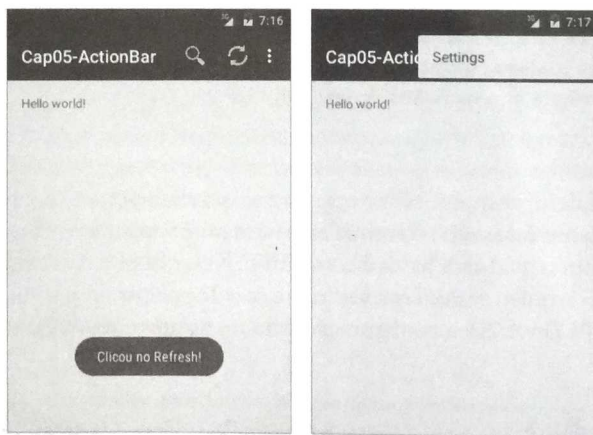


Figura 5.6 – Exemplo de action bar.

Os arquivos de menu são definidos no arquivo XML `/res/menu/menu_main.xml`. Como você pode ver no código-fonte a seguir, os botões na action bar aparecem na mesma ordem em que estão definidos no arquivo.

```
 /res/menu/menu_main.xml

<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" >
    <item android:id="@+id/action_search"
        android:icon="@drawable/ic_action_search" android:title="@string/action_search"
        android:showAsAction="always" />
    <item android:id="@+id/action_refresh"
        android:icon="@drawable/ic_action_refresh" android:title="@string/action_refresh"
        android:showAsAction="always" />
    <item android:id="@+id/action_settings"
        android:title="@string/action_settings"
        android:showAsAction="never" />
</menu>
```

Ao executar o projeto, os ícones de **Search** e **Refresh** vão aparecer na action bar, pois estão configurados com a opção `android:showAsAction="always"`. Já a ação de **Settings** fica no menu action overflow, pois está configurada com a opção `android:showAsAction="never"`. Ao clicar nos botões, o evento está sendo tratado. A figura 5.6 mostra o resultado ao clicar no botão de **Refresh**. Note que cada item de menu referencia um texto que está cadastrado no arquivo `/res/values/strings.xml`.

```
 /res/values/strings.xml

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">LivroAndroidCap5-ActionBar</string>
    <string name="hello_world">Hello world!</string>
    <string name="action_settings">Settings</string>
    <string name="action_search">Search</string>
    <string name="action_refresh">Refresh</string>
</resources>
```

Para utilizar esse arquivo XML de menu, a activity deve implementar o método `onOptionsItemSelected()` e inflar o menu XML. Isso é simples, conforme mostra este código:

```
@Override
public boolean onOptionsItemSelected(Menu menu) {
    // Infla o menu com os botões da action bar
```

```

        getMenuInflater().inflate(R.menu.menu_main, menu);
        return true;
    }

```

Uma vez que o menu com os botões da action bar estão configurados, basta implementar o método `onOptionsItemSelected()` para tratar os eventos gerados pelos botões. Por isso no arquivo XML de menu foi definido um id para cada item. O código-fonte completo da classe `MainActivity` do projeto pode ser visualizado a seguir.

MainActivity.java

```

...
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Infla o menu com os botões da action bar
        getMenuInflater().inflate(R.menu.menu_main, menu);
        return true;
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();
        if (id == R.id.action_search) {
            toast("Clicou no Search!");
            return true;
        } else if (id == R.id.action_refresh) {
            toast("Clicou no Refresh!");
            return true;
        } else if (id == R.id.action_settings) {
            toast("Clicou no Settings!");
            return true;
        }
        return super.onOptionsItemSelected(item);
    }
    private void toast(String msg) {
        Toast.makeText(this, msg, Toast.LENGTH_SHORT).show();
    }
}

```

Esse código está mostrando um alerta ao clicar em algum botão da action bar, portanto execute o projeto no emulador e configura o resultado. Conforme podemos verificar, adicionar os botões na action bar é realmente simples.

Nota: ao configurar um action button no arquivo XML de menu, é recomendável utilizar um ícone para deixar o visual mais interessante. Muitas vezes, somente de olhar o ícone, o usuário já sabe o que aquele botão faz, como são os casos da lupa de busca e o refresh. No entanto, mesmo que uma figura seja utilizada, é importante configurar o título, porque se o usuário pressionar o botão por dois segundos o Android vai mostrar um toast com o título desse botão. Isso também é importante por questões de acessibilidade, pois deficientes visuais utilizam softwares que conseguem ler tudo o que está escrito na tela. O mesmo conceito aplica-se ao componente `ImageView`, cujo atributo `android:contentDescription` fornece o texto explicativo da figura, o qual pode ser lido por estes softwares.

5.4 Opções de visualização dos action buttons (always, never, ifRoom)

Ao adicionar um action button na action bar, você deve escolher como será feita a visualização deste, isto é, se ele sempre ficará visível, ou se ficará disponível no menu do action overflow etc. Para isso, você pode utilizar qualquer uma destas constantes no atributo `app:showAsAction="xxx"`:

`always`

Indica que o botão sempre deve ficar visível como action button. É recomendado utilizar essa opção para definir as ações mais comuns do aplicativo.

`ifRoom`

Mostra o botão na action bar se existir espaço, ou move ele automaticamente para o menu action overflow caso não tenha. Muitas vezes essa é a configuração adequada para manter a compatibilidade com diversos tipos de dispositivos e também com telas na vertical e horizontal.

`withText`

Mostra o título do botão ao lado do ícone, caso tenha espaço disponível na action bar. Por exemplo, na horizontal existe mais espaço na action bar, portanto é possível exibir o título opcionalmente.

never

Nunca mostra o botão na action bar, deixando a ação no menu action overflow.

collapseActionView

Esse atributo indica que uma view que geralmente é grande deve ser contraída para exibir apenas o botão. Esse é o caso do botão de busca do Android, que fica contraído, mas ao ser clicado se expande para o usuário digitar o texto.

Com o tempo você entenderá melhor cada um desses itens, mas lembre-se de que geralmente é recomendado utilizar as opções `ifRoom` para garantir que se não houver espaço o botão apareça no menu action overflow, e a opção `never` se você tiver certeza que a opção deve ficar no menu action overflow. Também é possível combinar essas opções com o separador, como por exemplo: `app:showAsAction="ifRoom|withText"`. Agora é com você, brinque um pouco com essas opções e veja os resultados no emulador.

Nota: segundo as boas práticas do Android, os action buttons (botões com ações) devem ser utilizados para as ações mais comuns da tela, ou seja, as ações que serão mais utilizadas pelo usuário. Já as ações que são menos utilizadas, como por exemplo Ajuda e Configurações, devem ficar no menu action overflow (menu flutuante).

5.5 Template de ícones para os botões da action bar

Ao criar os ícones para os botões da action bar, é importante seguir o padrão de cores. Por exemplo, o projeto de exemplo está configurado para utilizar o tema Material conforme demonstrado a seguir:

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="AppTheme" parent="android:Theme.Material.Light.DarkActionBar">
    </style>
</resources>

```

Nesse caso, como foi configurada a action bar com o padrão escuro “Dark” os botões precisam ser brancos, por isso verifique que os ícones que coloquei na pasta `/res/drawable` são claros (Figura 5.7).

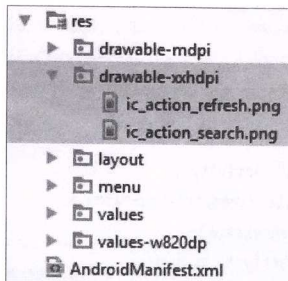


Figura 5.7 – Ícones para os botões da action bar.

Nota: os nomes dos ícones da action bar seguem a nomenclatura `ic_action_nome.png`.

Você deve estar se perguntando de onde eu tirei essas figuras. Felizmente existe uma coleção com o template de vários ícones com as ações mais comuns, como: atualizar, buscar, adicionar, deletar, copiar, colar, compartilhar etc. Para baixar o pacote de ícones da action bar, acesse a página da documentação da action bar e procure pelo link **Download the Action Bar Icon Pack**.

<http://developer.android.com/design/patterns/actionbar.html>

Ao descompactar o arquivo do pacote, você verá as imagens para o tema `Holo Dark` e `Holo Light`. No projeto que criei neste capítulo, estou utilizando o tema `Theme.Material.Light.DarkActionBar`. Isso significa que a action bar é escura e os botões devem ser brancos. Nesse caso o correto é selecionar algum ícone criado para o tema `Holo Dark`. Eu copiei as figuras de `Refresh` e `Search` para a pasta `/drawable/xhdpi`. No pacote de ícones você verá que existem as figuras para várias densidades, mas eu geralmente coloco no projeto apenas as figuras para a maior densidade, e deixo o Android redimensionar as imagens em tempo de execução, no caso de o aplicativo executar em telas com menor resolução.

5.6 Classe `android.app.ActionBar`

Por padrão, o título que aparece na action bar é o nome da activity configurado no atributo `android:label` da tag `<activity>` no arquivo `AndroidManifest.xml`.

Esse título pode ser customizado dinamicamente no código, e isso pode ser necessário dependendo da lógica da aplicação. A activity pode chamar o método `getActionBar()` para obter o objeto `android.app.ActionBar` e depois chamar o método `actionBar.setTitle("")` para customizar o título.



MainActivity.java

```
import android.app.Activity;
import android.app.ActionBar;
public class MainActivity extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ActionBar actionBar = getActionBar();
        actionBar.setTitle("Capítulo 5");
    }
    . . .
}
```

Ao executar esse exemplo no emulador, você verá que o título da action bar será “Capítulo 5”. Existem vários métodos na classe `android.app.ActionBar` e a lista a seguir contém alguns dos mais importantes.

`setCustomView(int ou View)`

Permite adicionar uma view customizada na action bar. Um exemplo de view customizada é o botão de busca do Android, que é encapsulado na classe `SearchView`.

`setTitle(string)`

Altera o título da action bar.

`setIcon(Drawable)`

Altera o ícone ‘home’ que por padrão mostra o ícone do projeto.

`setDisplayShowTitleEnabled(boolean)`

Configura se é para exibir o título na action bar.

`setDisplayShowHomeEnabled(boolean)`

Configura se é para exibir o logo/ícone na action bar, chamado de ‘home’.

`setDisplayHomeAsUpEnabled(boolean)`

Exibe a setinha para esquerda ‘up navigation’ para indicar que o usuário pode voltar à tela anterior ou navegar para cima na hierarquia de telas.

Embora existam muitos outros métodos na classe `android.app.ActionBar`, o recomendado é você ler a documentação oficial (javadoc) da classe. Alguns dos métodos são para criar a navegação por tabs e drop down, então vamos estudá-los depois.

5.7 SearchView

A action bar permite adicionar views customizadas, sendo que um exemplo clássico é o `SearchView`. Para este próximo exercício, vamos utilizar o `SearchView`; assim, quando o usuário tocar no botão de busca, o campo de texto será expandido para o usuário digitar o texto da busca. Felizmente o `SearchView` já faz todo esse trabalho e para utilizá-lo basta configurar a tag `android:actionViewClass` conforme demonstrado a seguir.



/res/menu/menu_main.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" >
    <item android:id="@+id/action_search"
        android:icon="@drawable/ic_action_search" android:title="@string/action_search"
        android:showAsAction="always"
        android:actionViewClass="android.widget.SearchView" />
    // . . . Outros botões de refresh e settings aqui
</menu>
```

Nota: o `SearchView` é uma view customizada que pode ser inserida na action bar, a qual é chamada de Action View. Um Action View tem por objetivo substituir o botão da action bar por alguma view customizada.

No código da activity, basta obter o objeto do `SearchView` e configurar para a aplicação tratar o evento `SearchView.OnQueryTextListener`. Note que o código a seguir está resumido, pois estou mostrando apenas o que precisa ser alterado referente ao `SearchView`.



MainActivity.java

```

...
import android.widget.SearchView;
public class MainActivity extends Activity {
    ...
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Infla o menu com os botões da action bar
        getMenuInflater().inflate(R.menu.menu_main, menu);
        MenuItem item = menu.findItem(R.id.action_search);
        SearchView searchView = (SearchView) item.getActionView();
        searchView.setOnQueryTextListener(onSearch());
        return true;
    }
    private SearchView.OnQueryTextListener onSearch() {
        return new SearchView.OnQueryTextListener(){
            @Override
            public boolean onQueryTextSubmit(String query) {
                // Usuário fez a busca
                toast("Buscar o texto: " + query);
                return false;
            }
            @Override
            public boolean onQueryTextChange(String newText) {
                // Mudou o texto digitado
                return false;
            }
        };
    }
    ...
}

```

Ao executar o projeto e clicar o ícone de busca, o `SearchView` vai expandir para o usuário digitar a busca. A figura 5.8 mostra o resultado em que eu digitei o texto Livro Android e pressionei <Enter> no emulador para fazer a busca.

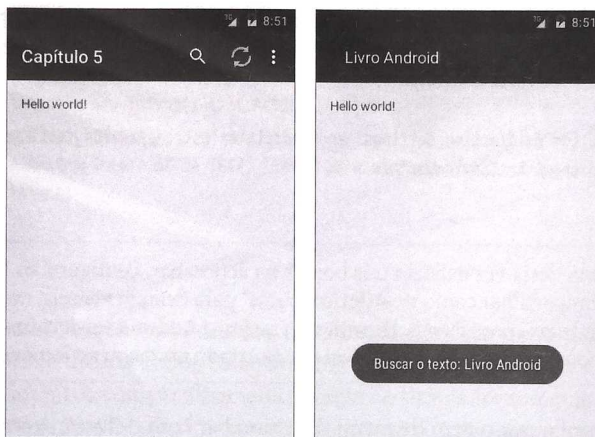


Figura 5.8 – Exemplo de SearchView.

5.8 Action provider

De forma similar ao action view, um action provider também substitui algum botão por um layout customizado. O action provider mais famoso de todos é o de compartilhar. Por exemplo, na galeria de fotos, ao compartilhar uma foto, o Android mostra automaticamente todos os aplicativos que podem enviar a foto, como por exemplo: Gmail ou Hangouts.

Para configurar um action provider em algum botão da action bar, basta adicionar a tag `android:actionProviderClass` e informar alguma subclasse de `android.view.ActionProvider`. Neste exemplo vamos utilizar a classe `android.widget.ShareActionProvider` que facilita o trabalho de compartilhar algum conteúdo.



/res/menu/menu_main.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools" >
    <item android:id="@+id/action_search"
        android:icon="@drawable/ic_action_search" android:title="@string/action_search"
        android:showAsAction="always" android:actionViewClass="android.widget.SearchView" />
    <item android:id="@+id/action_share"
        android:icon="@drawable/ic_action_share" android:title="@string/action_share"
        android:showAsAction="always"
        android:actionProviderClass="android.widget.ShareActionProvider" />
    <item
```

```

        android:id="@+id/action_refresh" android:icon="@drawable/ic_action_refresh"
        android:title="@string/action_refresh"
        android:showAsAction="ifRoom" />
    <item
        android:id="@+id/action_settings" android:title="@string/action_settings"
        android:showAsAction="never" />
</menu>

```

Nota: como desta vez existem três botões na action bar, configurei os botões de busca e compartilhar como `showAsAction="always"` para ficarem visíveis, mas o botão de refresh deixei como `showAsAction="ifRoom"`; assim, o Android vai decidir se o botão vai ficar como action button ou se vai ser mostrado no menu action overflow.

Depois de configurar o item de menu da action bar com o `ShareActionProvider`, basta customizar a `Intent` que será utilizada para disparar a mensagem ao sistema operacional, a fim de compartilhar o conteúdo. Vimos no capítulo 4, sobre a classe `Activity`, que uma `Intent` pode ser utilizada para navegar entre as telas da aplicação, mas na verdade uma `intent` representa uma mensagem que é enviada ao sistema operacional para os mais variados fins. No caso do compartilhamento, quando a `intent` é enviada, o sistema operacional vai perguntar para todas as aplicações instaladas quem é que consegue tratar essa mensagem para compartilhar o conteúdo. O código-fonte a seguir está resumido e mostra apenas o necessário para implementar o `ShareActionProvider`.



MainActivity.java

```

. . .
import android.widget.ShareActionProvider;
public class MainActivity extends Activity {
    . . .
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Infla o menu com os botões da action bar
        getMenuInflater().inflate(R.menu.menu_main, menu);
        // SearchView
        MenuItem searchItem = menu.findItem(R.id.action_search);
        SearchView searchView = (SearchView) searchItem.getActionView();
        searchView.setOnQueryTextListener(onSearch());

        // ShareActionProvider
        MenuItem shareItem = menu.findItem(R.id.action_share);
        ShareActionProvider share = (ShareActionProvider) shareItem.getActionProvider();
        share.setShareIntent(getDefaultIntent());
        return true;
    }
}

```

```

    }
    // Intent que define o conteúdo que será compartilhado
    private Intent getDefaultIntent() {
        Intent intent = new Intent(Intent.ACTION_SEND);
        intent.setType("text/*");
        intent.putExtra(Intent.EXTRA_TEXT, "Texto para compartilhar");
        return intent;
    }
    . . .
}

```

Ao executar a aplicação no emulador, o resultado deve ser como a figura 5.9. O `ShareActionProvider` mostra todos os aplicativos que conseguem tratar a mensagem enviada pela intent de compartilhamento, porém no simulador somente o aplicativo de mensagem está instalado. Em um dispositivo real, é comum encontrar vários aplicativos ao clicar no ícone de compartilhar. Nesse exemplo vimos a configuração de uma intent básica para compartilhar textos. Futuramente, durante o desenvolvimento do aplicativo para carros, veremos como compartilhar a foto de um carro.

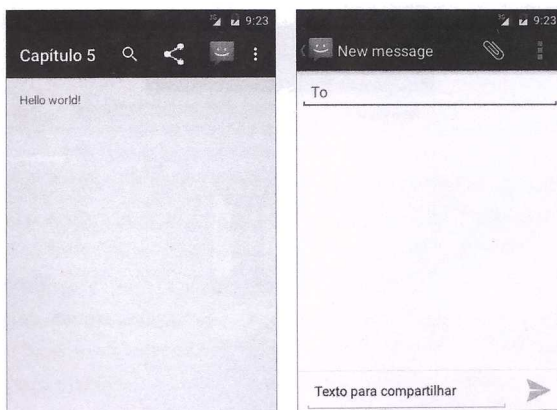


Figura 5.9 – Exemplo de `ShareActionProvider`.

5.9 Split action bar

Dependendo da aplicação, você pode precisar de mais espaço para exibir os botões na action bar, e uma opção é utilizar o split action bar, que vai deixar os botões na parte inferior da tela. Para habilitar a split action bar, edite o arquivo `AndroidManifest.xml` e insira o atributo `android:uiOptions="splitActionBarWhenNarrow"` e a tag `<meta-data>` na configuração da tag `<activity>`.

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <application ... >
        <activity android:name="br.livroandroid.cap5.actionbar.MainActivity"
            android:label="@string/app_name"
            android:uiOptions="splitActionBarWhenNarrow">
            <meta-data android:name="android.support.UI_OPTIONS"
                android:value="splitActionBarWhenNarrow" />
            <intent-filter>...</intent-filter>
        </activity>
    </application>
</manifest>
```

Feito isso, ao executar o projeto o resultado deve ser como a figura 5.10. Na verdade, a split action bar economiza espaço se necessário; caso o dispositivo esteja na horizontal, como existe espaço suficiente, os botões da action bar ficam na parte superior normalmente.



Figura 5.10 – Split action bar.

Nota: a configuração da split action bar deve ser feita activity por activity, e não é possível alterar pelo código, somente no arquivo *AndroidManifest.xml*. Essa configuração parece que foi descontinuada (deprecated) no Material Design, pois no Android 5.0 isso não funciona mais. O print que você está vendo é do Android 4.4 com o tema Holo.

5.10 Up navigation

No capítulo 4, sobre activity, criamos um aplicativo com uma tela de login com a mensagem de bem-vindo, a qual foi mostrada em outra tela. Para voltar à tela anterior, podemos utilizar o botão voltar nativo do Android, ou o up navigation, que é a seta que aponta para a esquerda e fica na action bar.

Como já explicado anteriormente, o up navigation é utilizado para subir na hierarquia de telas; seu funcionamento muitas vezes é parecido com o botão voltar, mas, dependendo do caso, pode ser diferente. Vimos que para mostrar a seta do up navigation na action bar basta chamar o método `getActionBar()`. `setDisplayHomeAsUpEnabled(true)` e depois o método `onOptionsItemSelected(MenuItem)` é chamado; o identificador `android.R.id.home` deve ser utilizado para identificar a ação.

```
@Override
public boolean onOptionsItemSelected(int featureId, MenuItem item) {
    // Clicou no "up navigation"
    if(item.getItemId() == android.R.id.home) {
        // O método finish() vai encerrar essa activity
        finish();
        return true;
    }
    return super.onOptionsItemSelected(featureId, item);
}
```

Se não quiser você não precisa implementar o método `onOptionsItemSelected(MenuItem)`, pois é possível configurar o *AndroidManifest.xml* para que o comportamento do up navigation da activity siga o padrão, que é voltar a tela anterior. Para isso existe o atributo `android:parentActivityName`, que permite configurar a activity mãe na hierarquia de telas, assim o Android sabe para qual tela precisa voltar. Como esse atributo foi criado no Android 4.0 (API Level 14), é preciso configurar também a tag meta-data para suportar as versões com Android 2.1 (API Level 7) ou superior.

Então faça o teste! Altere o código do arquivo *AndroidManifest.xml* do projeto que fizemos com a tela de login, e comente o método `onOptionsItemSelected(MenuItem)` na classe *BemVindoActivity*, pois ele não será mais necessário. Ao executar o projeto no emulador, o up navigation vai funcionar normalmente.



AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <application ... >
        <activity android:name=".MainActivity" android:label="@string/title_main_activity" >
```

```

    . . .
</activity>
<activity android:name=".BemVindoActivity" android:parentActivityName=".MainActivity" >
    <!-- Compatibilidade com a API level 7+ -->
    <meta-data android:name="android.support.PARENT_ACTIVITY"
        android:value=".MainActivity" />
</activity>
</application>
</manifest>

```

5.11 Navegação por tabs na action bar

A navegação por tabs na action bar permite ao aplicativo mostrar as principais seções que o usuário pode navegar e é considerada o nível (top-level) de navegação do seu aplicativo. As tabs devem ser utilizadas no caso de existirem poucas seções para navegar no aplicativo, sendo que o principal objetivo é mostrar ao usuário as seções que o aplicativo contém de forma simples e rápida.

A navegação por tabs é utilizada em aplicativos como o Google Play. Para aplicativos que tenham muitas seções de nível (top-level), um padrão de design que está sendo muito utilizado é o Navigation Drawer (menu lateral utilizado no aplicativo do Gmail), pois nele o menu lateral pode exibir diversas opções e mostrar uma rolagem na vertical.

Adicionar as tabs na action bar é bem simples, basta utilizar o método `setNavigationMode(ActionBar.NAVIGATION_MODE_TABS)` da classe `ActionBar` e depois criar as tabs com o método `addTab(x)`. Ao clicar numa tab, os métodos da interface `ActionBar.TabListener` são chamados para permitir que o aplicativo realize a ação necessária para trocar o conteúdo da tela. Para brincar com as tabs, vamos alterar o projeto `HelloActionBar` para criar três tabs. Ao clicar em cada tab, vamos apenas mostrar um alerta com o toast.



MainActivity.java

```

. . .
public class MainActivity extends Activity {
    . . .
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ActionBar actionBar = getActionBar();
        actionBar.setTitle("Capítulo 5");
    }
}

```

```

        // Configura a action bar para utilizar as tabs
        actionBar.setNavigationMode(ActionBar.NAVIGATION_MODE_TABS);
        // Cria as tabs (Passa como parâmetro o índice de cada tab: 1,2,3)
        actionBar.addTab(actionBar.newTab().setText("Tab 1")
            .setTabListener(new MyTabListener(this,1)));
        actionBar.addTab(actionBar.newTab().setText("Tab 2")
            .setTabListener(new MyTabListener(this,2)));
        actionBar.addTab(actionBar.newTab().setText("Tab 3")
            .setTabListener(new MyTabListener(this,3)));
    }
    . . .
}

```

Para o código compilar crie a classe `MyTabListener`, a qual implementa a interface `ActionBar.TabListener` e é responsável por tratar os eventos das tabs.



MyTabListener.java

```

. . .
import android.app.ActionBar;
import android.app.FragmentTransaction;
import android.content.Context;
import android.widget.Toast;

public class MyTabListener implements ActionBar.TabListener {
    private Context context;
    private int tabIndex;

    public MyTabListener(Context context, int tabIndex) {
        this.context = context;
        this.tabIndex = tabIndex;
    }

    @Override
    public void onTabSelected(ActionBar.Tab tab, FragmentTransaction ft) {
        // Chamado ao selecionar uma tab
        Toast.makeText(context, "Selecionou a tab: " + tabIndex, Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onTabUnselected(ActionBar.Tab tab, FragmentTransaction ft) {
        // Chamado quando a tab perde o foco (se outra tab é selecionada)
    }

    @Override
    public void onTabReselected(ActionBar.Tab tab, FragmentTransaction ft) {
        // Chamado quando uma tab é selecionada novamente.
    }
}

```

A figura 5.11 mostra o resultado desse exemplo com as três tabs criadas. Ao selecionar uma tab, o método `onTabSelected()` é chamado, o qual mostra com um toast (alerta) o índice da tab selecionada. Na prática, quando uma tab é selecionada, os aplicativos geralmente atualizam o conteúdo central da tela. Isso pode ser feito com a API de fragments que vamos estudar em outro capítulo. Um fragment é um tipo de view especial que pode ser inserido em determinado local da tela para gerenciar aquele conteúdo.

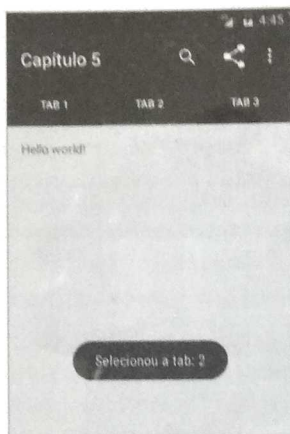


Figura 5.11 – Action bar com tabs.

Atenção: Você deve ter percebido que existem alguns alertas (warnings) no código das tabs. Isso acontece porque no Android 5.0 o modo de navegação por tabs na action bar foi descontinuado (deprecated).

Depois do Google I/O 2015, o Google disponibilizou uma biblioteca de componentes utilizados no Material Design chamada *Android Design Support Library*. Dentre os novos componentes, foi criado o *TabLayout*, que como o próprio nome já diz é utilizado para criar tabs. Vamos estudá-lo mais tarde ao desenvolver o aplicativo dos carros. Decidi mostrar como criar tabs com a action bar, mesmo que deprecated, pois este exemplo é um clássico e você precisa conhecê-lo. O motivo das tabs com action bar estar deprecated é porque o Google tem recomendado utilizar a *Toolbar* (Capítulo 1.2).

5.12 ActionBarCompat – a biblioteca de compatibilidade da action bar

No evento Google I/O 2013, o Google anunciou o suporte a action bar para versões anteriores do Android, compatível com o Android 2.1 (API Level 7) ou superior. Essa biblioteca é conhecida como biblioteca de compatibilidade v7 ou `ActionBarCompat`.

Para utilizar a action bar de compatibilidade, a primeira tarefa a fazer é declarar a dependência para a biblioteca v7 no arquivo *app/build.gradle* do módulo. Lembre-se de que existe o arquivo *build.gradle* que fica na raiz do projeto (não é esse) e o arquivo *app/build.gradle* que fica dentro do módulo. Você geralmente vai mexer no arquivo de configuração do módulo. Como exercício, vamos migrar o projeto de exemplo da action bar deste capítulo para utilizar a action bar de compatibilidade, para deixar o projeto compatível com o Android 2.1 (API Level 7).

Então vamos lá! Para começar a brincadeira, faça duas alterações no arquivo *app/build.gradle*.

1. Altere o atributo `minSdkVersion` para 7.
2. Adicione a dependência para a biblioteca `appcompat-v7`.



app/build.gradle

```
apply plugin: 'com.android.application'
android {
    compileSdkVersion 22
    buildToolsVersion "22.0.1"
    defaultConfig {
        applicationId "br.com.livroandroid.actionbar"
        minSdkVersion 7 // API Level 7 (Android 2.1)
        targetSdkVersion 22
        versionCode 1
        versionName "1.0"
    }
    . . .
}
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    // Dependência da biblioteca de compatibilidade v7
    compile 'com.android.support:appcompat-v7:22.1.1'
}
```

No item `defaultConfig` foi definido que a aplicação suporta o Android 2.1 (API Level 7) como a versão mínima e o target é o Android 5.1 (API Level 22). Lembre-se de que o target deve ser sempre a última versão e quando você estiver lendo este livro é

bem provável que existam versões novas. Na última linha do arquivo dentro do item `dependencies` é configurada a seguinte dependência:

```
compile 'com.android.support:appcompat-v7:22.1.1'
```

Isso indica ao Gradle que a biblioteca de compatibilidade deve ser adicionada como dependência do projeto, a qual contém a action bar de compatibilidade. No meu caso é a versão 22.1.1, pois é a versão do item `Android Support Library` que instalei pelo `SDK Manager`.

Nota: ao adicionar uma nova dependência no `build.gradle` clique no botão **Sync Now** que aparece no editor. Isso faz com que o Gradle baixe a dependência e a deixe ativa no projeto, para que as funcionalidades como o assistente de código e o compilador encontrem as classes desta biblioteca. Essa opção também pode ser acessada pelo menu `Tools > Android > Sync Project with Gradle Files`.

O simples fato de referenciar no arquivo `build.gradle` a biblioteca de compatibilidade da action bar faz com que o Gradle faça o build corretamente e resolva essa dependência na compilação do projeto. Mas lembre-se de clicar no botão **Sync Now** que aparece no editor!

Depois de adicionar a dependência, a primeira tarefa a fazer é configurar o tema do projeto no `AndroidManifest.xml`. O tema de compatibilidade é o `Theme.AppCompat` e suas variações são o `Theme.AppCompat.Light` e `Theme.AppCompat.Light.DarkActionBar`. Então altere o arquivo `/res/values/styles.xml` para utilizar o tema `Theme.AppCompat.Light.DarkActionBar`.

 `/res/values/styles.xml`

```
<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    </style>
</resources>
```

Como vamos utilizar o tema de compatibilidade `AppCompat`, não precisamos ter vários arquivos `styles.xml`, portanto apague os arquivos que estão em pastas `/res/values-v11`, `/res/values-v14`, `/res/values-v21` etc. O objetivo de utilizar essas pastas com os qualificadores por API Level é alterar o tema dependendo da versão do Android. Vamos utilizar o tema `AppCompat`, que é único para todas as versões, por isso podemos deixar apenas um arquivo `/res/values/styles.xml` no projeto.

Depois de configurar o tema, vamos verificar o que precisa ser alterado no código da activity. Primeiramente, altere o código da `MainActivity` para que ela seja filha

de `android.support.v7.app.AppCompatActivity`. Feito isso, em vez de utilizar o método `getActionBar()` que retorna a classe `android.app.ActionBar`, é obrigatório utilizar o método `getSupportActionBar()` que retorna a action bar de compatibilidade, cuja classe é `android.support.v7.app.ActionBar`.

O código a seguir mostra as alterações básicas da `MainActivity` para utilizar a action bar de compatibilidade:



MainActivity.java

```
import android.support.v7.app.AppCompatActivity;
import android.support.v7.app.ActionBar;
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ActionBar actionBar = getSupportActionBar();
        actionBar.setTitle("Capítulo 5");
        . . .
    }
    . . .
}
```

A `MainActivity` utiliza também o `SearchView` e o `ShareActionProvider`, e essas classes também fazem parte do pacote de compatibilidade, por isso precisamos alterar o código para utilizar as classes `android.support.v7.widget.SearchView` e `android.support.v7.widget.ShareActionProvider`. Além disso, é importante que ao configurar os itens de menu no método `onOptionsItemSelected(menu)` você utilize a classe `android.support.v4.view.MenuItemCompat` para obter o `SearchView` e o `ShareActionProvider`. A seguir podemos visualizar o código completo da classe `MainActivity` com todas as alterações necessárias para utilizar a action bar de compatibilidade. Preste atenção nos imports das classes, pois estamos importando os pacotes `android.support.v4` e `android.support.v7`.



MainActivity.java

```
. . .
import android.support.v4.view.MenuItemCompat;
import android.support.v7.app.ActionBar;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.SearchView;
import android.support.v7.widget.ShareActionProvider;
public class MainActivity extends AppCompatActivity {
    @Override
```



```

protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ActionBar actionBar = getSupportActionBar();
    actionBar.setTitle("ActionBar Compat");
    // Navegação por tabs
    actionBar.setNavigationMode(android.app.ActionBar.NAVIGATION_MODE_TABS);
    // Cria as tabs (Passa como parâmetro o índice de cada tab: 1,2,3)
    actionBar.addTab(actionBar.newTab().setText("Tab 1")
        .setTabListener(new MyTabListener(this,1)));
    actionBar.addTab(actionBar.newTab().setText("Tab 2")
        .setTabListener(new MyTabListener(this,2)));
    actionBar.addTab(actionBar.newTab().setText("Tab 3")
        .setTabListener(new MyTabListener(this,3)));
}
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // Infla o menu com os botões da action bar
    getMenuInflater().inflate(R.menu.menu_main, menu);
    // SearchView
    MenuItem searchItem = menu.findItem(R.id.action_search);
    SearchView searchView = (SearchView) MenuItemCompat.getActionView(searchItem);
    searchView.setOnQueryTextListener(onSearch());
    // ShareActionProvider
    MenuItem shareItem = menu.findItem(R.id.action_share);
    ShareActionProvider share = (ShareActionProvider)
        MenuItemCompat.getActionProvider(shareItem);
    share.setShareIntent(getDefaultIntent());
    return true;
}
. . .
}

```

Como o exemplo que fizemos utiliza a navegação por tabs, precisamos ajustar os imports da classe `MyTabListener` para utilizar a action bar de compatibilidade e também a versão de compatibilidade da classe `FragmentManager`. Vamos estudar mais detalhes sobre a classe `FragmentManager` no capítulo 8, sobre fragments.

`MyTabListener.java`

```

. . .
import android.support.v7.app.ActionBar;
import android.support.v4.app.FragmentManager;

```

```
public class MyTabListener implements ActionBar.TabListener {
    // Mesmo código aqui
}
```

Muito bem, estamos quase acabando. Para finalizar a migração para a biblioteca de compatibilidade, precisamos ajustar o arquivo XML de menu. No exemplo que fizemos utilizamos as tags `android:showAsAction`, `android:actionViewClass` e `android:actionProviderClass` para configurar os itens de menu e também o `SearchView` e o `ShareActionProvider`.

O problema é que as versões antigas do Android não conhecem essas tags, então precisamos adicionar o namespace `"app"` no XML e em vez de utilizar o atributo como `android:showAsAction="always"` vamos utilizar como `app:showAsAction="always"`.

A seguir podemos visualizar as alterações necessárias no arquivo de menu. Preste muita atenção em como o namespace `"app"` foi utilizado e veja que também estamos utilizando as classes `android.support.v7.widget.SearchView` e `android.support.v7.widget.ShareActionProvider` da biblioteca `v7`.



/res/menu/main.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:tools="http://schemas.android.com/tools"
      xmlns:app="http://schemas.android.com/apk/res-auto" tools:context=".MainActivity">
    <item android:id="@+id/action_search"
          android:icon="@drawable/ic_action_search" android:title="@string/action_search"
          app:showAsAction="always"
          app:actionViewClass="android.support.v7.widget.SearchView" />
    <item android:id="@+id/action_share"
          android:icon="@drawable/ic_action_share" android:title="@string/action_share"
          app:showAsAction="always"
          app:actionProviderClass="android.support.v7.widget.ShareActionProvider" />
    <item android:id="@+id/action_refresh"
          android:icon="@drawable/ic_action_refresh" android:title="@string/action_refresh"
          app:showAsAction="ifRoom" />
    <item android:id="@+id/action_settings"
          android:title="@string/action_settings"
          app:showAsAction="never" />
</menu>
```

Pronto! Isso é tudo. Relembrando, fizemos os seguintes passos para utilizar a biblioteca de compatibilidade da action bar:

1. Declarar a dependência no arquivo *app/build.gradle*.
2. A activity precisa ser filha de `android.support.v7.app.AppCompatActivity`.
3. No código da activity, utilize o método `getSupportActionBar()` para obter o objeto `android.support.v7.app.ActionBar`.
4. Ao configurar os itens de menu, utilize a classe `android.support.v4.view.MenuItemCompat`.
5. Utilize as classes de compatibilidade `android.support.v7.widget.SearchView` e `android.support.v7.widget.ShareActionProvider`.
6. No arquivo XML de menu, utilize o namespace "app" pois as versões antigas do Android não conhecem as tags `showAsAction`, `actionViewClass` e `actionProviderClass`.

Atualmente o Google recomenda utilizar as classes do pacote de compatibilidade, pois a vantagem é que essa biblioteca é compilada junto com o código do projeto, e a biblioteca é baixada pelo SDK Manager. Para qualquer melhoria ou correção de bugs nessas classes basta atualizar a biblioteca pelo SDK Manager. Já no caso de uma biblioteca nativa, ela está naturalmente embutida dentro do sistema operacional. Torna-se então uma boa prática utilizar sempre as bibliotecas de compatibilidade; isso vem funcionando tão bem que várias classes são distribuídas apenas nessas bibliotecas. É o caso do famoso componente `ViewPager` que ainda vamos estudar, que fica na lib v4 e para o qual nem existe uma versão nativa.

Dica: ao criar um projeto no Android Studio com suporte ao Android 2.3 (API Level 9), a action bar de compatibilidade será automaticamente configurada no projeto.

5.13 Links úteis

Para complementar sua leitura, seguem alguns links importantes da documentação oficial:

- **Android User Interface – Action Bar**

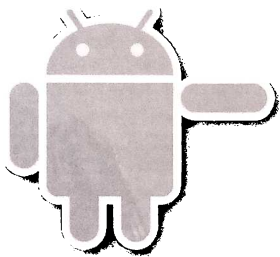
<http://developer.android.com/guide/topics/ui/actionbar.html>

- **Android Training – Adding the Action Bar**

<https://developer.android.com/training/basics/actionbar/index.html>

- **Android Training – Styling the Action Bar**

<https://developer.android.com/training/basics/actionbar/styling.html>



CAPÍTULO 6

Interface gráfica – gerenciadores de layout

No Android, existem diversos tipos de gerenciadores de layout. Alguns podem organizar os componentes na horizontal e vertical, outros podem organizar os componentes em uma tabela com linhas e colunas.

Neste capítulo, aprenderemos a organizar os componentes da tela com o layout desejado e construiremos diversos exemplos.

6.1 View

A classe `android.view.View` é a classe-mãe de todos os componentes visuais do Android, e suas diversas subclasses são utilizadas para criar a interface gráfica do aplicativo. Cada subclasse de `View` precisa implementar o método `onDraw(Canvas)`, responsável por desenhar o componente na tela.

Você deve ter percebido que muitas vezes, neste livro, chamamos uma `view` simplesmente de componente, para facilitar a leitura. Existem dois tipos de `views`/componentes, os chamados `widgets` e os gerenciadores de layout. Um `widget` é um componente simples que herda diretamente da classe `View`, como as classes `Button`, `ImageView` e `TextView`. Já os gerenciadores de layout consistem em subclasses de `android.view.ViewGroup` e são popularmente chamados apenas de `layouts`.

6.2 Classe ViewGroup

Um gerenciador de layout é utilizado para organizar a disposição dos componentes na tela. A classe-mãe de todos os gerenciadores de layout é a `android.view.ViewGroup`. Na figura 6.1, podemos visualizar a hierarquia das classes `View` e `ViewGroup`, com algumas de suas subclasses:

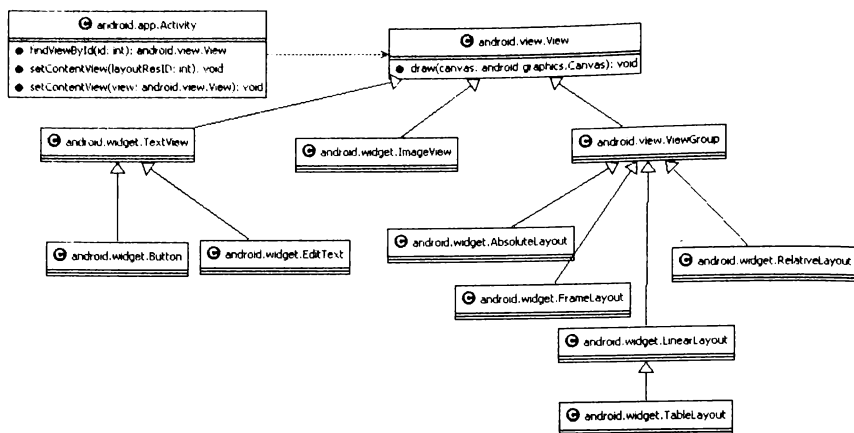


Figura 6.1 – Hierarquia da classe ViewGroup.

Conforme visualizado na figura 6.1, as principais classes de layout são:

Layout	Descrição
AbsoluteLayout	Permite posicionar os componentes, fornecendo as coordenadas x e y. Está deprecated.
FrameLayout	Tipo mais comum e simples de layout. Funciona como uma pilha, sendo que uma view fica por cima da outra.
LinearLayout	Utilizado para organizar os componentes na vertical ou horizontal.
TableLayout	É filho de LinearLayout e pode ser utilizado para organizar os componentes em uma tabela, com linhas e colunas.
RelativeLayout	Permite posicionar um componente relativo a outro, por exemplo, abaixo, acima ou ao lado de um componente já existente.

6.3 Configurando a altura e largura de uma view

Ao construir a interface gráfica da tela, é necessário configurar a largura e altura do layout principal, bem como a largura e altura de cada view inserida no layout. Isso é feito por meio dos parâmetros `android:layout_height` e `android:layout_width`.

Parâmetro	Descrição
<code>android:layout_height</code>	Especifica a altura de uma view.
<code>android:layout_width</code>	Especifica a largura de uma view.

O trecho de código a seguir mostra a configuração de largura e altura de um `LinearLayout`.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" >
```

Esses dois parâmetros podem receber os seguintes valores:

Valor	Descrição
<code>fill_parent</code>	Significa que o componente precisa ocupar todo o tamanho definido por seu pai (layout). Isso deve ser utilizado sempre que algum layout ou view precisa ocupar a tela inteira ou todo o espaço de layout definido pelo layout-pai.
<code>match_parent</code>	Idem ao <code>fill_parent</code> . Na verdade, o <code>fill_parent</code> virou deprecated no Android 2.2. Se você vai criar aplicações para Android 2.2 ou superior, poderá utilizar o <code>match_parent</code> . Ambas as notações têm o mesmo funcionamento, e o Google apenas descontinuou a notação <code>fill_parent</code> .
<code>wrap_content</code>	Utilizado para o componente ocupar apenas o tamanho necessário na tela, sem esticar.
número (dp)	Número inteiro especificando o tamanho, por exemplo, 50dp para ocupar 50 pixels da tela. A notação dp (density independent pixel) faz a conversão correta de pixels conforme a densidade/resolução da tela do dispositivo. Nunca utilize a notação 50px, sempre utilize 50dp para garantir que o aplicativo funcione bem em diversos tamanhos de telas.

Os parâmetros `android:layout_height` e `android:layout_width` são definidos pela classe interna `ViewGroup.LayoutParams`. Se você construir a tela utilizando diretamente a API Java, será necessário conhecer essa classe. Caso contrário, isso não é obrigatório.

Nos próximos tópicos veremos na prática como configurar a largura e altura do layout e de suas view, e explicaremos detalhadamente todos os gerenciadores de layout do Android.

6.4 Entendendo as constantes `wrap_content` e `match_parent`

Neste tópico, vamos criar alguns exemplos para você entender o significado das constantes `wrap_content` e `match_parent`, utilizadas para definir o tamanho do layout e das views.

Nota: para testar os exemplos crie o arquivo XML de layout no projeto, e faça a pré-visualização do layout no editor visual. Não será necessário executar o código. Ou, se preferir, abra o projeto deste capítulo no Android Studio e abra cada arquivo de layout no editor, assim ficará mais fácil para acompanhar as explicações.

Nesses arquivos de layout, vamos escolher o `FrameLayout` como tag raiz, mas isso na verdade não importa. O primeiro exemplo demonstra o `FrameLayout` utilizando os atributos `android:layout_width` e `android:layout_height` com o valor `wrap_content`. Isso significa que a altura e largura do layout será exatamente o espaço necessário para conter todos os componentes/views definidos na tela.

```
res/layout/exemplo_wrap_content.xml

<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:background="#8B8B8B" >
    <ImageView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:src="@drawable/android_green" />
</FrameLayout>
```

Nota: se a altura e largura de um layout forem especificadas como `wrap_content`, o tamanho desse layout será definido pelo tamanho necessário para desenhar todos os seus componentes.

Nesse exemplo, o `ImageView` utiliza o valor `wrap_content` para definir sua altura e largura, conseqüentemente a imagem ocupa somente o tamanho necessário. A figura 6.2 mostra a pré-visualização deste layout no editor.

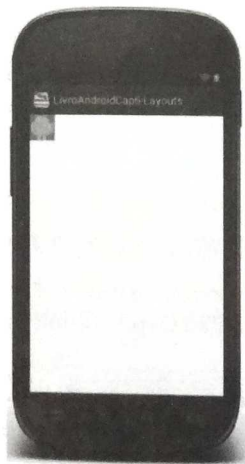


Figura 6.2 – Exemplo de `wrap_content`.

Observe que o fundo cinza atrás da imagem mostra o tamanho ocupado pelo layout principal, que nesse caso é o mesmo do tamanho da imagem. O restante da tela fica com fundo branco, pois é o tema padrão do projeto.

Nota: sempre que ficar em dúvida sobre qual o tamanho que o layout está ocupando, utilize o atributo `android:background` para definir uma cor para o fundo. Neste caso estou utilizando a cor cinza, assim é possível visualizar a área retangular ocupada pelo layout.

No próximo exemplo, vamos alterar a altura e largura do layout principal para `match_parent` para fazer com que ele preencha a tela inteira.

```
 /res/layout/exemplo_match_parent.xml

<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:background="#8B8B83" >
    <ImageView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:src="@drawable/android_green" />
</FrameLayout>
```

A figura 6.3 mostra que desta vez o layout com fundo cinza ocupou a tela inteira e a imagem ocupou apenas um pequeno espaço necessário para que fosse desenhada.

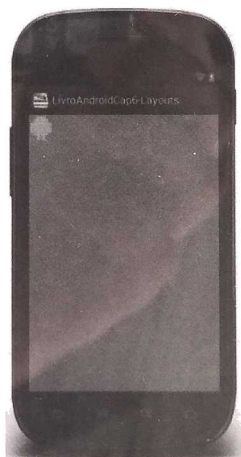


Figura 6.3 – Exemplo `match_parent`.

Como podemos ver o `fill_parent` ou `match_parent` esticam a view, e o `wrap_content` faz com que a view ocupe somente o espaço necessário.

Dica: um atalho interessante do Android Studio é o `Shift+Shift`, que abre uma janela para você digitar qualquer nome de arquivo para localizá-lo facilmente.

Agora vamos alterar também a largura e altura da imagem para `match_parent`. Isso significa que ela deve esticar e ocupar o tamanho ocupado por seu pai.



/res/layout/emplo_match_parent_imagem.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:background="#8B8B8B" >
    <ImageView
        android:layout_width="match_parent" android:layout_height="match_parent"
        android:src="@drawable/android_green" />
</FrameLayout>
```

O resultado dessa alteração pode ser visto na figura 6.4. Sempre que usar o valor `match_parent`, os componentes vão esticar/expandir para preencher o tamanho do layout-pai. Mas tome cuidado com imagens, pois geralmente elas devem ser definidas como `wrap_content`, justamente para não distorcer a figura.

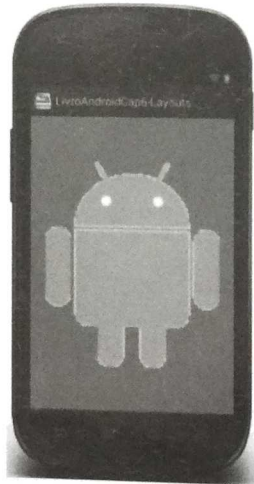


Figura 6.4 – Exemplo com a imagem esticando.

Em linhas gerais, é recomendado que a largura e altura do layout principal sejam definidas como `match_parent` para ocupar a tela inteira. Se necessário, configure a cor de fundo do layout para ter certeza do tamanho que ele está ocupando. Ao configurar a altura e largura para cada view da tela, é possível combinar os valores `match_parent` e `wrap_content` para expandir os componentes somente na vertical ou horizontal, conforme necessário.

Para você entender melhor como funciona o `match_parent` e `wrap_content`, vamos criar outro exemplo. Desta vez vamos utilizar o componente `EditText`, que representa um campo para o usuário digitar algo.



/res/layout/exemplo_textview_wrap_content.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="10dp" android:orientation="vertical" >
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/nome" />
    <EditText
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:inputType="text" />
    <Button
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="right" android:text="@string/ok" />
</LinearLayout>
```

A figura 6.5 mostra a pré-visualização desse layout. Observe que o campo de texto ocupou somente o espaço necessário, pois sua largura foi definida com `android:layout_width="wrap_content"` e não existe nenhum texto digitado. À medida que o usuário digitar algo, esse campo vai expandir automaticamente, o que pode não ser o desejado.

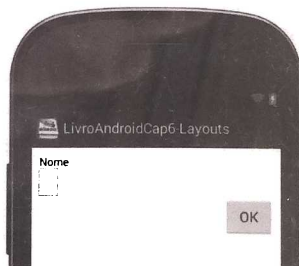


Figura 6.5 – Campo de texto com `wrap_content`.

Apenas por curiosidade, veja que o layout é um `LinearLayout` com a orientação vertical, o qual faz com que cada view seja adicionada logo abaixo da outra verticalmente. Outra curiosidade é o atributo `android:layout_gravity="right"` no botão, que faz com ele seja posicionado na direita.

Para corrigir o layout desse simples formulário, o ideal é aumentar a largura do campo de texto, o seja, a view `EditText`. Isso é feito com o atributo `android:layout_width="match_parent"` para que sua largura preencha todo o espaço disponível do layout-pai.

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="10dp" android:orientation="vertical" >
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/nome" />
    <EditText
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:inputType="text" />
    <Button
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="right" android:text="@string/ok" />
</LinearLayout>

```

A figura 6.6 mostra a pré-visualização do layout, e dessa vez o campo de texto esticou na largura para ocupar todo o tamanho disponível. Observe que ele apenas esticou na largura, pois não mexemos na configuração de altura, a qual continua com `wrap_content`. Como exercício, você pode esticar o campo de texto na altura para ver o que acontece.

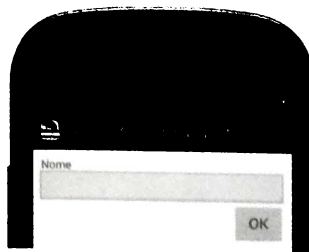


Figura 6.6 – Campo de texto com `match_content` na largura.

Note que não adicionamos valores fixos para o tamanho, pois é recomendado criar telas que se ajustam automaticamente conforme a resolução. Por isso não importa se o aplicativo vai executar no smartphone, tablet ou numa gigante TV que o layout será o mesmo.

6.5 Framelayout

A classe `android.widget.FrameLayout` é o mais simples de todos os gerenciadores de layout e é utilizada para empilhar uma view sobre outra.

É possível adicionar vários componentes dentro do `FrameLayout`, e sempre os últimos componentes ficarão sobre os anteriores, seguindo o conceito de uma pilha, em que o último elemento fica no topo. O caso mais comum disso é para inserir um `ProgressBar` por cima de alguma outra view. A propósito, a classe `ProgressBar` é uma view que desenha aquela bolinha que fica girando para simular algum processamento.

Para exemplificar, vamos inserir um `ProgressBar` por cima do botão **OK** do formulário do exemplo anterior. Dessa forma, ao clicar no botão podemos disparar a animação do `ProgressBar` para simular o processamento de alguma tarefa. Vamos aprender como fazer isso no código depois, por enquanto vamos estudar apenas os layouts.



/res/layout/exemplo_frame_layout_1.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" android:padding="10dp" >
    <TextView android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/nome" />
    <EditText android:layout_width="match_parent" android:layout_height="wrap_content"
        android:inputType="text" />
    <FrameLayout android:layout_width="wrap_content" android:layout_height="wrap_content" >
        <Button
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:layout_gravity="right" android:text="@string/ok" />
        <ProgressBar android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:layout_gravity="center" />
    </FrameLayout>
</LinearLayout>
```

A figura 6.7 mostra a pré-visualização desse layout. Talvez na figura não fique visível, mas o `ProgressBar` é o círculo animado que está sobre o botão. É justamente isso que o `FrameLayout` faz, ele adiciona uma view sobre a outra, como uma pilha.



Figura 6.7 – *FrameLayout com um ProgressBar.*

Nota: vamos estudar mais sobre os componentes visuais no próximo capítulo; no momento preocupe-se apenas com os gerenciadores de layout. Apenas por curiosidade, o `ProgressBar` ao ser inserido no layout vai exibir a animação automaticamente. Para parar a animação, você deve esconder o `ProgressBar` com o método `setVisible(boolean)` definido na classe `android.view.View`.

Outro caso comum de utilização do `FrameLayout` com o `ProgressBar` é quando temos uma lista na tela, como por exemplo a lista de contatos da agenda. Como boa prática de programação, é recomendado que, durante o carregamento dos dados da lista, o `ProgressBar` seja utilizado para exibir uma animação para o usuário. O seguinte layout mostra um `ListView`, que é o componente que exibe uma lista e no centro um `ProgressBar` para mostrar a animação.

 /res/layout/exemplo_frame_layout_2.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" android:padding="10dp" >
    <ListView
        android:id="@+id/listView"
        android:layout_width="match_parent" android:layout_height="match_parent" />
    <ProgressBar
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="center" />
</FrameLayout>
```

A figura 6.8 mostra a pré-visualização desse layout. O editor mostra uma lista para simular que o `ListView` está preenchido. Já o `ProgressBar` é utilizado para mostrar a animação para o usuário, enquanto os dados da lista estão sendo carregados. O `ProgressBar` está por cima do `ListView`, pois é isso que o `FrameLayout` faz.

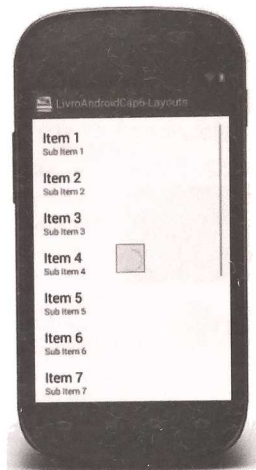


Figura 6.8 – *FrameLayout com um ListView e ProgressBar.*

6.6 LinearLayout

A classe `android.widget.LinearLayout` é um dos gerenciadores de layout mais utilizados, sendo possível organizar os componentes na horizontal (padrão) ou na vertical. A orientação, nesse caso, é configurada pelo atributo `android:orientation`.

A seguir temos um exemplo de como utilizar a classe `LinearLayout`.



/res/layout/exemplo_linear_layout_1.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="horizontal ou vertical"
    android:padding="10dp" >
    <ImageView android:src="@drawable/android_blue"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:contentDescription="@string/content_description" />
    <ImageView android:src="@drawable/android_green"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:contentDescription="@string/content_description" />
</LinearLayout>
```

A figura 6.9 exibe a pré-visualização desse layout, primeiro com a configuração `android:orientation="horizontal"` e ao lado com a configuração `android:orientation="vertical"`.

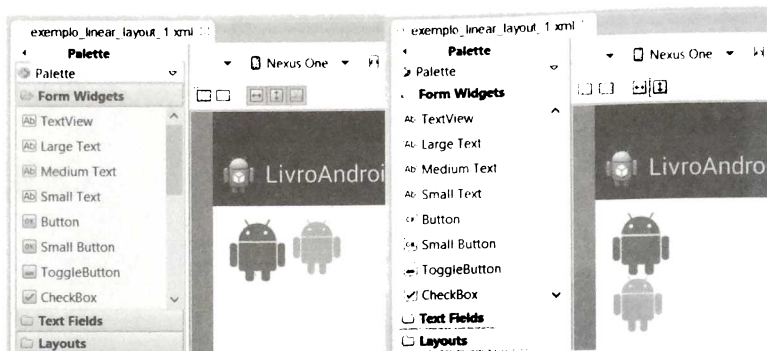


Figura 6.9 – `LinearLayout` na horizontal e vertical.

O funcionamento do `LinearLayout` é bem simples e não tem muito o que explicar. Apenas lembre-se de que, caso o atributo `android:orientation` seja omitido, o padrão é horizontal.

6.7 `LinearLayout` – controle do alinhamento “`layout_gravity`”

Agora demonstraremos como alinhar os componentes de forma centralizada, na esquerda ou direita da tela. Para isso, usaremos o atributo `android:layout_gravity`. Os valores válidos para esse atributo são `top` (cima), `bottom` (baixo), `left` (esquerda), `right` (direita), `center_vertical`, `fill_vertical`, `center_horizontal`, `fill_horizontal`, `center` e `fill`.

Anteriormente, no exemplo do formulário com o `FrameLayout` e o `ProgressBar`, utilizamos o seguinte código:

```
<ProgressBar
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:layout_gravity="center" />
```

Repare que o `ProgressBar` está declarado com o atributo `android:layout_gravity="center"`, o qual faz com que ele seja posicionado no centro do seu layout-pai. Para demonstrar outras maneiras de utilizar o atributo `android:layout_gravity`, criaremos uma tela com três imagens inseridas na vertical, que ficam alinhadas na esquerda, no centro e na direita da tela.

 /res/layout/exemplo_linear_layout_2_gravity.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" >
    <ImageView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="left" android:src="@drawable/android_green" />
    <ImageView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="center" android:src="@drawable/android_blue" />
    <ImageView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="right" android:src="@drawable/android_green" />
</LinearLayout>
```

A figura 6.10 exibe as imagens alinhadas à esquerda, no centro e na direita conforme esperado.



Figura 6.10 – Controlando o alinhamento com o atributo `layout_gravity`.

6.8 LinearLayout – controle do peso

Outra forma de organizar os elementos na tela é atribuir um peso (porcentagem) para cada um. O componente com o maior peso ocupará o maior espaço na tela. Para isso podemos utilizar o atributo `android:layout_weight`.

Para demonstrar isso, criaremos um exemplo com um formulário, que contém os campos nome, email e observações. Nesse exemplo o campo observações deve ocupar a área restante da tela para que o usuário possa digitar um texto um pouco maior. Por isso foi definido o peso = 1 para esse componente utilizando o atributo `android:layout_weight="1"`.


```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent" android:layout_height="match_parent" >
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Nome" />
    <EditText
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:textColor="#ff0000" />
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Email" />
    <EditText
        android:layout_width="match_parent" android:layout_height="wrap_content" />
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Observações" />
    <EditText
        android:layout_width="match_parent" android:layout_height="0dp"
        android:layout_weight="1" />
    <Button
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Enviar" />
</LinearLayout>

```

A figura 6.11 mostra a pré-visualização desse layout. Observe que o campo observações ocupou o restante da tela. Isso ocorreu porque o campo observações foi definido com o atributo `android:layout_weight="1"` e `android:layout_height="0dp"`. Como ele é o único componente que tem peso, esticou na altura.

Nota: sempre que utilizar o peso com o atributo `android:layout_weight`, deve-se definir a largura ou altura da view com `0dp`. Essa notação indica que a largura ou altura da view deve respeitar o peso atribuído.

Sempre que apenas um componente na tela precisar ocupar o tamanho restante, basta atribuir o valor 1 para o atributo `android:layout_weight`, como acabamos de verificar. Mas o que acontece se for necessário expandir mais de um componente? Para facilitar a explicação criaremos um novo exemplo, com três campos de texto.

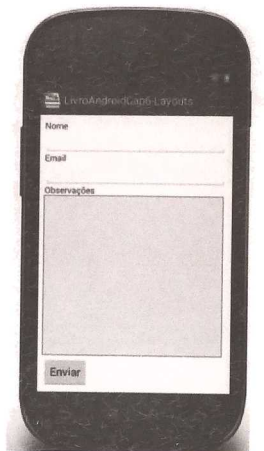


Figura 6.11 – Exemplo de `layout_weight` com peso = 1.



/res/layout/exemplo_linear_layout_4_weight.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent" android:layout_height="match_parent" >
    <EditText android:layout_weight="1"
        android:layout_width="match_parent" android:layout_height="0dip"
        android:text="Texto (weight=1)" />
    <EditText android:layout_weight="2"
        android:layout_width="match_parent"
        android:layout_height="0dip"
        android:text="Texto (weight=2)" />
    <EditText android:layout_weight="3"
        android:layout_width="match_parent" android:layout_height="0dip"
        android:text="Texto (weight=3)" />
</LinearLayout>
```

Observe que cada `EditText` define um valor diferente para o atributo `android:layout_weight`. O primeiro define o valor 1, o segundo o valor 2, e o terceiro o valor 3. A figura 6.12 mostra a pré-visualização desse layout. Como podemos verificar, o campo de texto com o maior peso ocupou a maior área da tela.

Lembre-se de que também a altura de cada componente foi definida com `0dp` ou `0dip`, que é a mesma coisa. Isso permite deixar a altura configurável pelo peso que esse componente tem. O mesmo conceito também se aplica à largura.



Figura 6.12 – Exemplo do atributo *layout_weight* para controlar o peso.

Para entender melhor o que aconteceu, vamos fazer uma superconta para somar estes valores: $1 + 2 + 3 = 6$. Como a soma dos valores do atributo *layout_weight* é 6, cada campo de texto vai ocupar o espaço proporcional na tela, relativo ao total, como se 6 fosse 100%. Assim, podemos visualizar que o último campo definido como *layout_weight=3* ocupou metade da tela, sendo que 3 é 50% de 6.

Para entender melhor como a atribuição de pesos funciona, vamos brincar um pouco com esse exemplo, alterando o código-fonte do arquivo XML de layout. Por exemplo, vamos alterar o primeiro campo de texto para *peso = 3*. Nesse caso, agora a soma ficaria $3 + 2 + 3 = 8$ (Figura 6.13). Como o primeiro e o terceiro campo têm os mesmos pesos, eles ocuparão o mesmo espaço na tela. Já o campo do meio será o menor.

Para finalizar este tópico, vou dar uma dica muito importante e que muitos desenvolvedores Android demoram a descobrir ou entender. O componente *ListView* que é utilizado para criar listas deve ser utilizado na maioria das vezes com *peso = 1* na vertical para esticar a sua altura.

```
<ListView
    android:id="@+id/listView"
    android:layout_width="match_parent"
    android:layout_height="0dp" android:layout_weight="1" />
```

A diferença é que, se configurarmos a altura como `android:layout_height="match_parent"`, a lista vai esticar sem pensar em quem está abaixo dela, mandando todas as outras views para fora da tela. Mas com a configuração do peso, o `ListView` vai esticar até onde puder, pois geralmente somente ele tem o peso = 1, ou seja, somente o `ListView` deve esticar. À medida que formos avançando na leitura deste livro, vamos aprender mais detalhes e voltamos a revisar este conceito.

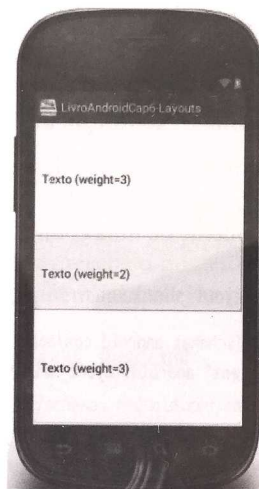


Figura 6.13 – Exemplo do atributo `layout_weight` para controlar o peso.

6.9 `TableLayout` – uso de uma tabela com linhas e colunas

A classe `android.widget.TableLayout` é um dos layouts mais úteis para construir algumas telas, como por exemplo formulários. Cada linha da tabela é formada por um `android.widget.TableRow`, que é uma subclasse de `LinearLayout` e consequentemente pode conter outros componentes, em que cada um representa uma linha na tabela.

O `TableLayout` tem os atributos `android:stretchColumns` e `android:shrinkColumns`, os quais recebem os índices das colunas, separadas por vírgula, que devem ser alteradas. Por exemplo, pode-se informar para o atributo `android:stretchColumns` os valores "2,3", para alterar apenas as colunas de índice 2 e 3. Observe que o índice inicia em zero, de modo que nesse caso as colunas 3 e 4 serão alteradas.

Veja a seguir a definição de cada atributo:

- `android:stretchColumns` – Faz com que as colunas ocupem o espaço disponível na tela, expandindo-as. Use esse atributo quando for necessário que uma coluna ocupe a linha inteira. O funcionamento é como um `colspan` de uma página HTML.
- `android:shrinkColumns` – Faz com que as colunas especificadas sejam sempre exibidas na tela. Caso o valor do texto seja muito grande e fique para fora da tela, a linha é quebrada e o texto é exibido em várias linhas na mesma coluna

6.10 TableLayout e shrinkColumns – contração de colunas

O atributo `android:shrinkColumns` recebe o número das colunas que é preciso contrair, ou seja, força o conteúdo da coluna para ficar dentro da tabela. Esse recurso deve ser utilizado quando o componente de uma coluna é longo demais.

 `/res/layout/exemplo_table_layout_shrink.xml`

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:shrinkColumns="2">
    <TableRow>
        <TextView android:text="Coluna 1" />
        <TextView android:text="Coluna 2" />
    </TableRow>
    <TableRow>
        <TextView android:text="Coluna 1" />
        <TextView android:text="Coluna 2" />
        <TextView android:text="Texto Grande que vai sair da tela mas o shrinkColumns=2 foi
            utilizado para quebrar." />
    </TableRow>
    <TableRow>
        <TextView android:text="Coluna 1" />
        <TextView android:text="Coluna 2" />
        <TextView android:text="Coluna 3" />
    </TableRow>
</TableLayout>
```

Neste exemplo, a coluna 2 da segunda linha (lembre-se de que o índice da coluna começa em 0) tem um texto muito grande. O atributo `android:shrinkColumns` foi usado para forçar seu conteúdo a ser exibido na tela (Figura 6.14). Dessa forma, o texto foi quebrado e consequentemente foi necessário utilizar mais de uma linha.

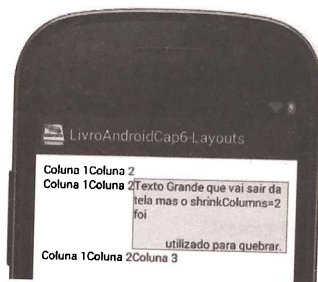


Figura 6.14 – TableLayout com shrinkColumns.

6.11 TableLayout e stretchColumns – expansão de colunas

O atributo `android:stretchColumns` recebe o número das colunas que precisam ser expandidas (forçar o preenchimento da tela inteira).

 /res/layout/exemplo_table_layout_stretch.xml

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:stretchColumns="1" >
    <TextView android:text="@string/lista_de_produtos" />
    <View
        android:layout_height="2dip"
        android:background="#FF909090" />
    <TableRow>
        <TextView android:text="@string/produto_a" />
        <TextView
            android:layout_gravity="right"
            android:text="@string/realis_100" />
    </TableRow>
    <TableRow>
        <TextView android:text="@string/produto_b" />
        <TextView
            android:layout_gravity="right"
            android:text="@string/realis_200" />
    </TableRow>
    <TableRow>
        <TextView android:text="@string/produto_c" />
        <TextView
```

```

        android:layout_gravity="right"
        android:text="@string/realis_300" />
    </TableRow>
    <View
        android:layout_height="2dip"
        android:background="#FF909090" />
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:gravity="bottom|center_horizontal" >
        <Button
            android:layout_width="wrap_content"
            android:layout_height="35dip"
            android:text="@string/cancelar" />
        <Button
            android:layout_width="wrap_content"
            android:layout_height="35dip"
            android:text="@string/comprar" />
    </LinearLayout>
</TableLayout>

```

A pré-visualização desse layout pode ser vista na figura 6.15.



Figura 6.15 – *TableLayout* com *stretchColumns*.

Observe que o tamanho da segunda coluna é maior que o da primeira justamente pelo atributo `android:stretchColumns` estar configurado. Veja também que o atributo

`android:layout_gravity="right"` é usado para posicionar os valores da segunda coluna à direita da tabela. Os botões na parte inferior da tela estão centralizados, pois o `LinearLayout` está com a configuração e `android:gravity="bottom|center_horizontal"`.

Também é possível adicionar outros componentes (views) na tabela, não somente o `TableRow`. Caso outro componente seja adicionado na tabela ele ocupará a linha inteira. Veja que esse exemplo demonstrou a integração entre dois layouts diferentes, `LinearLayout` e `TableLayout`.

Nota: um gerenciador de layout também é um tipo de `View` e por isso é possível inserir um layout dentro de outro para criar interfaces mais complexas. Esse conceito é chamado de layouts aninhados.

6.12 TableLayout – criando um formulário

Um formulário com o `TableLayout` é similar a um formulário HTML. Cada `TableRow` representa uma linha, e cada componente, uma coluna. Neste exemplo a segunda coluna foi configurada para expandir seu conteúdo, utilizando o atributo `android:stretchColumns="1"` (lembre-se de que o índice começa em 0). Dessa forma, os campos do login e senha da segunda coluna ocuparão todo o espaço restante da linha.

 /res/layout/exemplo_table_layout_form.xml

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:stretchColumns="1">
    <TableRow>
        <TextView android:text="Usuário: " android:padding="3dip" />
        <EditText android:id="@+id/campoLogin" android:padding="3dip" />
    </TableRow>
    <TableRow>
        <TextView android:text="Senha: " />
        <EditText android:id="@+id/campoSenha" android:inputType="textPassword" />
    </TableRow>
    <TableRow android:gravity="right">
        <Button android:id="@+id/login" android:text="Login" />
    </TableRow>
</TableLayout>
```


Observe que a última linha usa o atributo `android:gravity="right"` para alinhar o botão de login à direita. O campo de senha contém o atributo `android:inputType="textPassword"` para mostrar os `***` no lugar do texto (Figura 6.16).



Figura 6.16 – *TableLayout* para a construção de um formulário.

6.13 GridLayout

A classe `android.widget.GridLayout` organiza as views em linhas e colunas; para isso a view deve utilizar os atributos `layout_row` e `layout_column`.

No exemplo a seguir vamos criar quatro botões. Veja que o layout foi definido com 2 linhas e 2 colunas e cada view é inserida em determinada posição.



/res/layout/exemplo_grid_layout.xml

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="16dp"
    android:columnCount="2" android:rowCount="2">
    <Button
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_column="0" android:layout_row="0"
        android:text="Botão 1" />
    <Button
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_column="1" android:layout_row="0"
        android:text="Botão 2" />
```

```

<Button
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:layout_column="0" android:layout_row="1"
    android:text="Botão 3" />
<Button
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:layout_column="1" android:layout_row="1"
    android:text="Botão 4" />
</GridLayout>

```

O GridLayout também contém alguns atributos que não usei nesse exemplo, como os atributos `android:layout_columnSpan` e `android:layout_rowSpan`, os quais podem ser utilizados para que a view ocupe mais de uma coluna ou linha, de forma similar ao `span` em páginas HTML. Você também pode usar o atributo `android:layout_gravity` para informar se a view deve esticar na vertical, horizontal ou ambas (`fill_vertical`, `fill_horizontal` e `fill`). O resultado do exemplo pode ser visualizado na figura 6.17.

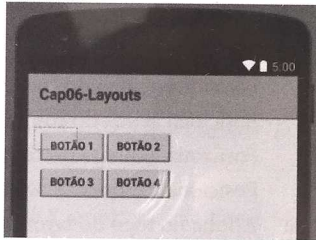


Figura 6.17 – GridLayout organizando as views em linhas e colunas.

É importante ressaltar que o GridLayout só está disponível a partir do Android 4.0 (API Level 14), mas felizmente podemos utilizar a biblioteca de compatibilidade compatível com o Android 2.1 (API Level 7). Para isso basta adicionar a seguinte dependência no arquivo `app/build.gradle`:

```
compile 'com.android.support:gridlayout-v7:21+'
```

Feito isso, podemos utilizar a classe `android.support.v7.widget.GridLayout`. Observe que por questões de compatibilidade, é obrigatório utilizar o prefixo `app` nos atributos do layout, pois eles não são reconhecidos nas versões anteriores do Android.

 `/res/layout/exemplo_grid_layout.xml`

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.GridLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent" android:layout_height="match_parent"

```

```

    android:padding="16dp"
    app:columnCount="2" app:rowCount="2">
    <Button
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Botão 1"
        app:layout_column="0" app:layout_row="0" />
    <!-- Outros botões aqui -->
</android.support.v7.widget.GridLayout>

```

6.14 RelativeLayout

A classe `android.widget.RelativeLayout` pode posicionar os componentes ao lado, abaixo ou acima de outro componente já existente. Para isso é necessário definir um id para cada componente da tela, pois o posicionamento de um componente depende de outro. Os seguintes atributos podem ser utilizados para informar a posição do novo componente inserido relativo ao componente já existente:

Atributo	Descrição
<code>android:layout:below</code>	Posiciona abaixo do componente indicado.
<code>android:layout:above</code>	Posiciona acima do componente indicado.
<code>android:layout:toRightOf</code>	Posiciona à direita do componente indicado.
<code>android:layout:toLeftOf</code>	Posiciona à esquerda do componente indicado.
<code>android:layout_alignParentTop</code>	Alinha no topo do layout-pai.
<code>android:layout_alignParentBottom</code>	Alinha abaixo do layout-pai.
<code>android:layout_alignParentRight</code>	Alinha à direita do layout-pai.
<code>android:layout_alignParentLeft</code>	Alinha à esquerda do layout-pai.
<code>android:layout_alignTop</code>	Alinha no topo do componente indicado.
<code>android:layout_alignBottom</code>	Alinha abaixo do componente indicado.
<code>android:layout_alignRight</code>	Alinha à direita do componente indicado.
<code>android:layout_alignLeft</code>	Alinha à esquerda do componente indicado.
<code>android:layout_marginTop</code>	Utilizado para definir um espaço na margem superior do componente.
<code>android:layout_marginBottom</code>	Utilizado para definir um espaço na margem inferior do componente.
<code>android:layout_marginRight</code>	Utilizado para definir um espaço à direita do componente.
<code>android:layout_marginLeft</code>	Utilizado para definir um espaço à esquerda do componente.

A seguir podemos visualizar um exemplo para construir um formulário:

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="wrap_content"
    android:background="#8B8B83" android:padding="16dp">
    <TextView android:id="@+id/labelUsuario"
        android:layout_width="55dip" android:layout_height="wrap_content"
        android:text="@string/usuario"/>
    <!-- Ao lado do label Usuário "layout_toRight" -->
    <EditText android:id="@+id/campoUsuario"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:background="@android:drawable/editbox_background"
        android:layout_toRightOf="@id/labelUsuario"
        android:inputType="text"/>
    <!-- Senha: abaixo do campoUsuario "layout_below", alinhado a esquerda -->
    <TextView android:id="@+id/labelSenha"
        android:layout_width="55dip" android:layout_height="wrap_content"
        android:layout_below="@id/campoUsuario"
        android:gravity="left" android:text="@string/senha"/>
    <!-- Ao lado do label Senha "layout_toRight" -->
    <EditText android:id="@+id/campoSenha"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:background="@android:drawable/editbox_background"
        android:layout_toRightOf="@id/labelSenha"
        android:layout_alignTop="@id/labelSenha"
        android:inputType="textPassword" />
    <!-- Abaixo do campoSenha "layout_below", alinhado a direita -->
    <Button android:id="@+id/btLogin" android:layout_marginTop="10dp"
        android:layout_width="wrap_content" android:layout_height="35dip"
        android:layout_below="@id/campoSenha"
        android:layout_alignParentRight="true"
        android:text="@string/login" />
</RelativeLayout>

```

A pré-visualização desse arquivo pode ser vista na figura 6.18.

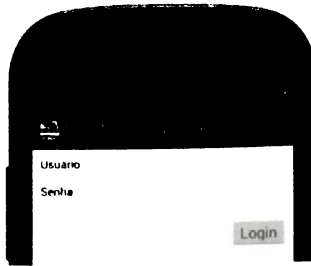


Figura 6.18 – RelativeLayout para criar um formulário.

Considerações sobre o exemplo:

1. O label Usuário é o primeiro componente, justamente por isso, é posicionado acima, alinhado à esquerda.
2. O campo de texto Usuário foi alinhado à direita do label Usuário, com o atributo `android:layout_toRightOf="@id/labelUsuario"`. Observe que o campo de texto utiliza o atributo `layout_width="match_parent"` para preencher o restante da tela.
3. O label Senha foi alinhado abaixo do campo Usuário, com o atributo `android:layout_below="@id/campoUsuario"`. Observe que o campo Usuário foi utilizado como referência, não o label Usuário. Isso foi feito porque a altura do campo é maior do que a do label, sendo assim o campo Usuário era um ponto de referência melhor. Como fica à direita, foi necessário alinhar o label Senha à esquerda para voltar à margem da página. Isso foi feito com o atributo `android:gravity="left"`.
4. O campo de texto Senha foi alinhado à direita do label Senha.
5. O botão de Login foi inserido abaixo do campo de Senha, com o atributo `android:layout_below`. Para alinhar à direita foi utilizado o atributo `android:layout_alignParentRight="true"`, relativo ao campo Senha. Para deixar um pequeno espaço (margem) entre a linha do campo Senha e o início do botão, foi utilizado o atributo `android:layout_marginTop="10dp"` com o valor de 10dp. Também existem os atributos `android:layout_marginLeft` e `android:layout_marginRight` com funcionamento semelhante.

O `RelativeLayout` é bem flexível ao construir as telas, permitindo que um componente seja inserido, sempre relativo a outro já existente. Mas isso pode também aumentar a complexidade da construção da tela, uma vez que é necessário dominar todos os atributos utilizados para controlar as posições em que os componentes são inseridos. Uma desvantagem desse gerenciador de layout é que, se a localização de um componente for alterada, pode quebrar o layout de toda a tela, pois o posicionamento de um componente depende de outro.

Esse layout é adorado por alguns desenvolvedores e criticado por outros, então caberá a você decidir usá-lo ou não. Saiba que o Google recomenda esse layout, pois é possível criar com menos código as mesmas telas que se fariam com muitas linhas aninhando vários `LinearLayout` com `vertical` e `horizontal`. Atualmente o editor visual é bem eficiente ao fazer o `drag and drop` dos componentes e gerar automaticamente todos esses atributos do `RelativeLayout`; portanto, faça o teste no editor visual.

6.15 `AbsoluteLayout` (deprecated)

A classe `android.widget.AbsoluteLayout` permite controlar exatamente a posição dos componentes, fornecendo suas coordenadas `x` e `y`, utilizando os atributos `android:layout_x` e `android:layout_y`.

O `AbsoluteLayout` foi descontinuado (deprecated), pois o layout da tela pode ficar diferente ou totalmente errado em dispositivos com diferentes resoluções de tela. Isso acontece porque 100 pixels em uma tela pode ser uma coisa, mas em outro dispositivo pode ser outra, tudo depende da densidade e resolução da tela de cada dispositivo. Este breve tópico serve apenas para avisá-lo para nunca utilizar o `AbsoluteLayout`, portanto, nenhum exemplo será demonstrado.

6.16 Utilizando layouts aninhados para criar telas complexas

Neste próximo exemplo vamos demonstrar como utilizar layouts aninhados. Digamos que precisamos construir um formulário, em que cada campo apareça logo abaixo do outro na vertical. Contudo, na última linha é necessário exibir dois botões: **cancelar** e **login**, mas eles precisam ser exibidos lado a lado.



/res/layout/exemplo_linear_layout_form_aninhado.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent" android:layout_height="match_parent" >
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/nome" />
    <EditText
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:textColor="#ff0000" android:inputType="text"/>
    <TextView
```

```

        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/senha" />
<EditText android:inputType="textPassword"
        android:layout_width="match_parent" android:layout_height="wrap_content" />
<LinearLayout android:orientation="horizontal"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:background="#cccccc" android:gravity="center">
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/cancelar" />
    <Button android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/login" />
</LinearLayout>
</LinearLayout>

```

A pré-visualização desse layout pode ser vista na figura 6.19. O `LinearLayout` principal da tela é vertical, mas no final foi utilizado um `LinearLayout` horizontal para conter os dois botões. Observe que o layout que contém os dois botões foi definido com uma cor de fundo cinza para demonstrar seu tamanho.



Figura 6.19 – *LinearLayout* aninhado.

6.17 Criação de um layout pela API – `LinearLayout`

Conforme já estudamos, para criar telas no Android é possível definir os layouts em arquivos XML, ou criar toda a tela usando a API Java, o que na maioria dos casos não é recomendado.

A seguir, será fornecido um exemplo de criação de um formulário utilizando a classe `LinearLayout` somente com a API Java. Para isso basta criar uma instância de `LinearLayout` e chamar o método `setContentView(view)`. Assim como no XML, ao criar o layout pela API também é obrigatório definir os atributos `layout_width` (largura) e `layout_height` (altura), que geralmente recebem os valores `match_parent`

e `wrap_content`. Esses parâmetros podem ser informados utilizando a classe `android.widget.LinearLayout.LayoutParams`, que por sua vez recebe no construtor as constantes `LayoutParams.MATCH_PARENT` e `LayoutParams.WRAP_CONTENT`.

Para adicionar uma `View` dentro do layout, é utilizado o método `addView(view)`, que recebe uma subclasse de `android.view.View`, como `TextView`, `EditText`, `ImageView`, `Button` etc. Para testar essa activity, abra o projeto deste capítulo no Android Studio e execute no emulador.



ExemploLinearLayoutAPIActivity.java

```
import android.widget.LinearLayout;
import android.widget.LinearLayout.LayoutParams;
public class ExemploLinearLayoutAPIActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Cria o layout
        LinearLayout layout = new LinearLayout(this);
        layout.setOrientation(LinearLayout.VERTICAL);
        layout.setLayoutParams(new LayoutParams(LayoutParams.MATCH_PARENT,
            LayoutParams.MATCH_PARENT));
        layout.setPadding(10, 10, 10, 10); // Espaçamento em pixels
        TextView nome = new TextView(this);
        nome.setText("Nome:");
        nome.setLayoutParams(new LayoutParams(LayoutParams.WRAP_CONTENT,
            LayoutParams.WRAP_CONTENT));
        layout.addView(nome);
        EditText tnome = new EditText(this);
        tnome.setLayoutParams(new LayoutParams(LayoutParams.MATCH_PARENT,
            LayoutParams.WRAP_CONTENT));
        layout.addView(tnome);

        // Focus
        tnome.requestFocus();
        TextView senha = new TextView(this);
        senha.setText("Senha:");
        senha.setLayoutParams(new LayoutParams(LayoutParams.WRAP_CONTENT,
            LayoutParams.WRAP_CONTENT));
        layout.addView(senha);
        EditText tsenha = new EditText(this);
        tsenha.setLayoutParams(new LayoutParams(LayoutParams.MATCH_PARENT,
            LayoutParams.WRAP_CONTENT));
        layout.addView(tsenha);
    }
}
```



```

        // Botão alinhado a direita
        Button ok = new Button(this);
        ok.setLayoutParams(new LayoutParams(LayoutParams.WRAP_CONTENT,
            LayoutParams.WRAP_CONTENT));
        ok.setGravity(Gravity.RIGHT);
        ok.setText("OK");
        layout.addView(ok);

        // Informa o layout que foi criado pela API
        setContentView(layout);
    }
}

```

Como executar esse exemplo vou deixar por sua conta, pois a única coisa que ele faz é criar um simples formulário. Mas saiba que é possível criar todas as telas utilizando somente a API: basta usar as subclasses de `View` desejadas. Os métodos das classes são semelhantes (quase sempre) aos atributos do XML. Por exemplo, a classe `TextView` tem o método `setText(x)`, que corresponde ao atributo `android:text` do XML. Fica a seu critério escolher a maneira mais adequada para criar a interface gráfica. Entretanto, criar o layout em XML é a forma recomendada pelo Google, pois separa a interface gráfica da lógica de negócios.

Nota: para testar os próximos exemplos, recomendo abrir o projeto deste capítulo no Android Studio e executar no emulador. A `MainActivity` desse projeto vai mostrar uma lista com cada exemplo que vamos estudar.

6.18 Criação de um layout pela API – `TableLayout`

O próximo exemplo de código mostra como criar um layout pela API com a classe `TableLayout`. Observe que para cada linha é criado um `TableRow`.



Exemplo `TableLayoutAPIActivity.java`

```

import android.widget.TableLayout;
import android.widget.TableLayout.LayoutParams;
public class ExemploTableLayoutAPIActivity extends Activity {
    @Override
    protected void onCreate(Bundle icle) {
        super.onCreate(icle);
    }
}

```

```

        // Cria o layout
        TableLayout tabela = new TableLayout(this);
        tabela.setLayoutParams(new LayoutParams(LayoutParams.MATCH_PARENT,
            LayoutParams.MATCH_PARENT));
        // Expande a coluna 1
        tabela.setColumnStretchable(1, true);

        // Linha 1
        TableRow linha1 = new TableRow(this);
        TextView nome = new TextView(this);
        nome.setText("Nome:");
        linha1.addView(nome);
        EditText tnome = new EditText(this);
        // Focus no campo nome
        tnome.requestFocus();
        linha1.addView(tnome);

        // Linha 2
        TableRow linha2 = new TableRow(this);
        TextView senha = new TextView(this);
        senha.setText("Senha:");
        linha2.addView(senha);
        EditText tsenha = new EditText(this);
        tsenha.setTransformationMethod(new PasswordTransformationMethod());
        linha2.addView(tsenha);

        // Linha 3
        TableRow linha3 = new TableRow(this);
        linha3.setGravity(Gravity.RIGHT);
        // Botão alinhado à direita
        Button ok = new Button(this);
        ok.setText(" Login ");
        linha3.addView(ok);

        // Adiciona as linhas
        tabela.addView(linha1);
        tabela.addView(linha2);
        tabela.addView(linha3);

        // Informa o layout
        setContentView(tabela);
    }
}

```

Como exercício, deixo para você executar este exemplo no emulador.

6.19 ScrollView

Quando existem muitos elementos na tela, surge a necessidade de fazer a rolagem (scroll). Para isso, podemos utilizar a classe `android.widget.ScrollView`. A classe `ScrollView` deve conter apenas um componente-filho e conforme o seu tamanho ele fará a rolagem ou não. Portanto, deve-se adicionar um layout dentro do `ScrollView`, como por exemplo o `LinearLayout`, que por sua vez pode receber outros componentes. A seguir demonstraremos a utilização do `ScrollView`:



/res/layout/activity_exemplo_scrollview.xml

```
<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="wrap_content">
    <LinearLayout android:id="@+id/layout1"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:orientation="vertical">
        <!-- 100 TextViews serão adicionados dinamicamente aqui -->
    </LinearLayout>
</ScrollView>
```

Observe que o `ScrollView` contém um `LinearLayout`. Para simular a barra de rolagem, será criada uma activity que vai obter o `LinearLayout` definido pelo id `layout1` e vai adicionar dinamicamente pelo código Java vários `TextView` na tela. Como o `LinearLayout` foi definido com o sentido vertical, cada `TextView` ficará embaixo do outro.



ExemploScrollViewActivity.java

```
public class ExemploScrollViewActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getActionBar().setDisplayHomeAsUpEnabled(true);
        setContentView(R.layout.activity_exemplo_scrollview);
        LinearLayout layout = (LinearLayout) findViewById(R.id.layout1);
        for (int i = 0; i < 100; i++) {
            TextView text = new TextView(this);
            // É obrigatório o layout_width e layout_height
            text.setLayoutParams(new LayoutParams(LayoutParams.WRAP_CONTENT,
                LayoutParams.WRAP_CONTENT));
            text.setText("Texto: " + i);
            layout.addView(text);
        }
    }
}
```

A figura 6.20 mostra o resultado do exemplo, em que podemos ver que foi criada a barra de rolagem devido à grande lista de elementos adicionados no layout. Esse exemplo também é interessante, uma vez que mostra como recuperar o `LinearLayout` do arquivo XML e adicionar várias views dinamicamente pela API. No próximo tópico vou explicar algo importante sobre a action bar e navegação de telas, portanto preste atenção nestas duas figuras. Você deve ter percebido que a primeira figura é referente ao projeto de exemplo deste capítulo, no qual a `MainActivity` mostra todos os exemplos na lista.

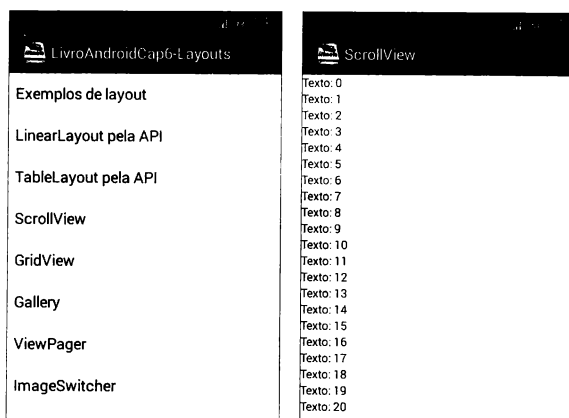


Figura 6.20 – Exemplo de `ScrollView`.

Dica: caso precise fazer a rolagem na horizontal, utilize o `HorizontalScrollView`.

6.20 Alguns detalhes sobre a `ActionBar` e o “up navigation”

Na figura do exemplo anterior, o título da action bar da primeira tela é o nome do projeto, mas o título da segunda activity é `ScrollView`.

Para entender a explicação deste código, veja o código-fonte do projeto de exemplo deste capítulo.

Isso acontece porque no `AndroidManifest.xml` eu cadastrei um label para a activity, e esse label é utilizado como o título da action bar. Caso o label não esteja definido é utilizado o nome do projeto.

```
<activity android:name=".ExemploScrollViewActivity" android:label="@string/scrollview"
    android:parentActivityName=".MainActivity"/>
```

Observe também que, ao declarar a `activity`, eu informei a `activity`-pai, que é a `MainActivity` do projeto. Isso é feito com a tag `android:parentActivityName=".MainActivity"`. Por isso ao entrar no exemplo do `ScrollView` você pode clicar no botão de “up navigation”, que é a setinha para esquerda na action bar, e automaticamente o sistema vai voltar para a tela anterior.

Lembrando que, conforme explicado no capítulo 5, sobre action bar, para habilitar o botão “up navigation” basta utilizar a seguinte linha de código:

```
getActionBar().setDisplayHomeAsUpEnabled(true);  
// Ou com a biblioteca de compatibilidade v7  
getSupportActionBar().setDisplayHomeAsUpEnabled(true);
```

6.21 LayoutInflater – inflando arquivos XML

No exemplo anterior, vimos como adicionar views dinamicamente no `ScrollView`. Para isso foi preciso instanciar no código objetos do tipo `TextView`. Como já foi explicado anteriormente, o recomendado é sempre fazer o layout em XML. Então como fazer nos casos em que é preciso criar views dinamicamente?

Para isso podemos criar um arquivo XML de layout com apenas a parte da view que precisamos adicionar, que neste caso é um `TextView` em cada linha do `ScrollView`. Então podemos ter um arquivo XML de layout como este:

```
 /res/layout/inflate_textview.xml  
  
<TextView xmlns:android="http://schemas.android.com/apk/res/android"  
    android:id="@+id/textView"  
    android:layout_width="wrap_content" android:layout_height="wrap_content"  
>
```

Note que este arquivo de layout contém o `TextView` que estávamos instanciando por programação no exemplo anterior. Desta vez esse mesmo `TextView` será instanciado diretamente por meio do arquivo XML, procedimento que é conhecido pelos desenvolvedores Android como “inflar um XML”. Isso é feito com a classe `LayoutInflater`, conforme demonstrado a seguir:

```
 ExemploScrollViewActivity.java
```

```
public class ExemploScrollViewActivity extends Activity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);
```

```
getActionBar().setDisplayHomeAsUpEnabled(true);
setContentView(R.layout.activity_exemplo_scrollview);
LinearLayout layout = (LinearLayout) findViewById(R.id.layout1);
for (int i = 0; i < 100; i++) {
    // Cria o TextView inflando o arquivo de layout, "infla o xml".
    LayoutInflater inflater = LayoutInflater.from(this);
    TextView text = (TextView) inflater.inflate(R.layout.inflate_textview, layout, false);
    // Agora basta usar a view inflada normalmente.
    text.setText("Texto: " + i);
    layout.addView(text);
}
}
```

Nota: é recomendável não criar o layout no código Java, mesmo que sejam poucas partes. Se for necessário criar um objeto do tipo `View` no código, utilize o `LayoutInflater` para “inflar” um arquivo XML. Lembre-se de que nem sempre a view inflada será tão simples quanto neste exemplo. No arquivo XML podemos definir um layout bem complexo, o que justifica utilizar um arquivo XML para manter o código organizado.

Os parâmetros do método `inflate(resource, root, attachToRoot)` são:

Valor	Descrição
<code>resource</code>	Arquivo XML com o layout que precisa ser inflado. O retorno será o objeto <code>View</code> desse layout.
<code>root</code>	Layout “container” no qual esta view será adicionada. Deve ser informado para a view conhecer o tamanho e posicionamento do layout-pai.
<code>attachToRoot</code>	Flag que indica se a view deve ser adicionada no layout automaticamente. É recomendado informar <code>false</code> e chamar o método <code>addView(view)</code> manualmente para adicionar a view no layout.

6.22 Links úteis

Para complementar sua leitura, segue o link da documentação oficial:

- **Android API Guides – Layouts**

<http://developer.android.com/guide/topics/ui/declaring-layout.html>



CAPÍTULO 7

Interface gráfica – View

A classe `android.view.View` é a classe-mãe de todos os componentes visuais do Android. Este capítulo explica várias de suas subclasses. No início, explicaremos como trabalhar com recursos de texto, cores e imagens do Android. Depois, demonstraremos diversos componentes disponíveis na API para criar interfaces visuais ricas.

Também aprenderemos a criar uma subclasse de `View` e desenhar com a API de Canvas, além de criar exemplos interessantes, como movimentar objetos e imagens pela tela.

7.1 Arquivo `/res/values/strings.xml`

O arquivo `/res/values/strings.xml` contém as mensagens de texto do projeto, para fazer a internacionalização. Um conceito importante sobre os arquivos XML com os recursos do projeto é que eles podem ter qualquer nome. Se você quiser criar o arquivo `/res/values/mensagens.xml` em vez de `/res/values/strings.xml`, é perfeitamente possível. O Android sabe o tipo do arquivo pelo seu conteúdo; assim, no caso de arquivos de mensagens, eles devem sempre iniciar com a tag `<resources>` e conter várias tags `<string>` com a chave e o valor do texto.

A seguir, temos alguns textos que vamos utilizar nos próximos exemplos. Para continuar, crie o projeto `Demo-Views`, ou se preferir abra o projeto de exemplo deste capítulo.



`/res/values/strings.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">LivroAndroidCap7-View</string>
    <string name="hello_world">Hello world!</string>
    <string name="action_settings">Settings</string>
```

```

<string name="msg_verde_e_branco">Texto verde e branco </string>
<string name="msg_azul_e_branco">Texto azul e branco</string>
<string name="msg_vermelho_e_branco">Texto vermelho e branco</string>
<string name="texto1">Texto campo 1 azul</string>
</resources>

```

Para acessar essas mensagens no código Java, utilize a classe `R` com a sintaxe `R.string.chave_mensagem`.

```

TextView t = (TextView) findViewById(R.id.text);
t.setText(R.string.msg_verde_e_branco);

```

Já no caso de um arquivo XML, utilize a sintaxe `@string/chave_mensagem`.

```

<TextView android:text="@string/msg_verde_e_branco" . . . />

```

Para traduzir os textos para diferentes idiomas, basta criar uma pasta `/res/values-idioma`.

/res/values/values-pt/strings.xml

Caso queira deixar claro que esta pasta é para o português do Brasil, adicione a região.

/res/values/values-pt-rBR/strings.xml

7.2 Arquivo XML com as cores

Da mesma forma que criamos um arquivo para conter as mensagens, podemos criar um arquivo com as cores. Esse arquivo também pode ter qualquer nome, mas deve ter um formato específico. Cada tag `<color>` tem o nome da cor e o código RGB.

 */res/values/colors.xml*

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="vermelho">#ff0000</color>
    <color name="azul">#0000ff</color>
    <color name="verde">#00ff00</color>
    <color name="branco">#ffffff</color>
</resources>

```

Se o código RGB tiver seis caracteres, são as cores normais: vermelho, verde e azul. Mas, se tiver oito caracteres, a primeira dupla é referente à transparência. Por exemplo, o código a seguir define uma cor azul.

```

<color name="azul">#0000ff</color>

```


Em seguida, temos a cor azul com 50% de transparência.

```
<color name="azul">#500000ff</color>
```

Para utilizar uma cor no código, utilize o método `getResources().getColor(cor)`.

```
TextView t = (TextView) findViewById(R.id.text);
t.setTextColor(getResources().getColor(R.color.branco));
t.setBackgroundColor(getResources().getColor(R.color.verde));
t.setText(R.string.msg_verde_e_branco);
```

Para utilizar as cores no arquivo XML, utilize a sintaxe `@color/cor`. Podemos visualizar a seguir um `TextView` que mostra como deixar o texto branco com a cor de fundo verde.

```
<TextView android:layout_width="match_parent" android:layout_height="wrap_content"
    android:text="@string/msg_verde_e_branco"
    android:textColor="@color/branco" android:background="@color/verde" />
```

Observe que nesse exemplo utilizamos os atributos `android:background` e `android:textColor` do `TextView` para definir a cor de fundo e a cor do texto. Para o valor da cor foi usado um recurso definido no arquivo `/res/values/colors.xml`, mas também poderíamos ter usado a cor em RGB diretamente no código.

```
<TextView android:layout_width="match_parent" android:layout_height="wrap_content"
    android:text="@string/msg_verde_e_branco"
    android:textColor="#ffffff" android:background="#00ff00" />
```

De qualquer forma, é recomendado deixar as cores separadas nos arquivos de recursos, pois dessa forma a cor pode ser alterada de forma centralizada, facilitando a manutenção do código.

7.3 Arquivo XML para criar um estilo CSS

Já ouviu falar de CSS em páginas para internet? No Android, existe algo parecido e são chamados de estilos. Um tema que estudamos anteriormente é um conjunto de estilos.

Com um arquivo de estilo, podemos definir de uma só vez determinado padrão de cor, assim como o tipo da fonte como negrito e itálico. Dessa forma, é possível dar um nome a esse estilo e usá-lo no código. Um arquivo de estilos também pode ter qualquer nome, desde que seja formado pelas tags `<style name="nomeEstilo">` e a tag `<item>` para compor cada elemento.

`/res/values/css.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="estiloExemplo">
        <item name="android:textSize">14sp</item>
        <item name="android:textColor">#ffffff</item>
        <item name="android:background">#ff0000</item>
        <item name="android:textStyle">italic</item>
    </style>
</resources>
```

Para referenciar o estilo no código, utilize a classe `R` com a constante `R.style.nome_estilo`.

```
TextView t = (TextView) findViewById(R.id.text);
t.setTextAppearance(this, R.style.estiloExemplo);
```

Para referenciar o estilo no arquivo XML, utilize a sintaxe `@style/nome_estilo`.

```
<TextView android:text="@string/msg_vermelho_e_branco" style="@style/estiloExemplo" ... />
```

7.4 Exemplo completo com estilos

Agora que estudamos como criar arquivos com os recursos de texto, cores e estilos, criaremos uma tela com três campos de texto para demonstrar como utilizar os recursos.

Se você quiser, é possível criar uma *activity* com o seguinte código para testar os exemplos. No entanto, como vou utilizar os arquivos XML de layout para demonstrar, a própria pré-visualização do editor já é suficiente para entender os exemplos.

`ExemploTextoCoresActivity.java`

```
public class ExemploTextoCoresActivity extends Activity {
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_exemplo_texto_cores);
    }
}
```

A seguir, podemos visualizar o código-fonte do arquivo de layout da *activity*.

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" android:padding="16dp" >
    <!-- Direito as cores -->
    <TextView android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="@string/msg_azul_e_branco"
        android:background="@color/azul" android:textColor="@color/branco"
        android:textStyle="bold" />
    <TextView
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:text="@string/msg_verde_e_branco"
        android:background="@color/verde" android:textColor="@color/branco"
        android:textStyle="bold" />
    <!-- Texto com @style -->
    <TextView style="@style/estiloExemplo"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:text="@string/msg_vermelho_e_branco" />
</LinearLayout>

```

Nesse código, são utilizadas as sintaxes `@string/chave_msg`, `@color/nome_cor` e `@style/nome_estilo` para acessar as mensagens, cores e estilos definidos nos arquivos de recursos. Na pré-visualização do layout, podemos ver uma tela com três textos coloridos, conforme a figura 7.1. Devido à impressão, não é possível visualizar claramente as cores no livro; portanto, abra o arquivo no editor.

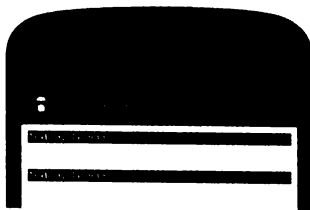


Figura 7.1 – Estilos de fonte e cor.

Nota: os estilos são utilizados em todos os componentes visuais (views) e não apenas para definir a cor dos textos. Mas isso é um pouco mais avançado e você vai aprender com o tempo. Você se lembra dos temas *Holo* e *Material*? Cada tema é um conjunto de estilos que é aplicado em cada view.

7.5 View – A classe responsável por desenhar elementos na tela

A classe `android.view.View` é utilizada como base para qualquer componente gráfico, e toda subclasse de `View` precisa implementar o método `onDraw(Canvas canvas)`, o qual é responsável por desenhar os elementos. Na lista a seguir, podemos verificar alguns dos principais métodos da classe `View`.

Atributo	Descrição
<code>requestFocus()</code>	Solicita o foco do componente. Para isso, os métodos <code>isFocusable()</code> ou <code>isFocusableInTouchMode()</code> precisam retornar <code>true</code> .
<code>setPadding(esquerda, cima, direita, baixo)</code>	Informa o espaço em pixels que deve ser inserido à esquerda, direita, acima e abaixo do componente antes de mostrar o seu conteúdo. Isso é muito utilizado em layouts que fazem <code>padding</code> para dar uma pequena margem antes de inserir as <code>views</code> filhas. O atributo correspondente no XML é <code>android:padding</code> , o qual define automaticamente o mesmo valor para todos os parâmetros. Caso seja necessário, no XML, informar corretamente o <code>padding</code> usado para a esquerda, direita, acima e abaixo da <code>view</code> , podem-se utilizar os atributos <code>android:paddingLeft</code> , <code>android:paddingRight</code> , <code>android:paddingTop</code> e <code>android:paddingBottom</code> , respectivamente.
<code>setVisibility(v)</code>	Pode receber três valores, definidos pelas constantes <code>View.VISIBLE</code> , <code>View.INVISIBLE</code> e <code>View.GONE</code> . Ambas as constantes <code>INVISIBLE</code> e <code>GONE</code> não mostram a <code>View</code> na tela. A diferença é que o <code>INVISIBLE</code> não mostra a <code>view</code> , mas deixa o espaço que ela ocuparia reservado (em branco). Já a constante <code>GONE</code> literalmente remove a <code>view</code> da tela. O atributo no XML correspondente a esse método é o <code>android:visibility</code> , o qual recebe os seguintes valores: <code>visible</code> , <code>invisible</code> e <code>gone</code> .
<code>requestLayout()</code>	Solicita ao Android para refazer o layout da tela. Será visto com mais detalhes posteriormente, quando criarmos um exemplo com uma.
<code>invalidate()</code>	Invalida a <code>View</code> , solicitando ao Android para desenhar a <code>view</code> novamente. Será visto com mais detalhes posteriormente quando criarmos nossa própria <code>View</code> customizada.
<code>onSizeChanged(int largura, int altura, int larguraAntiga, int alturaAntiga)</code>	Chamado pelo Android sempre que um componente altera seu tamanho, informando as novas largura e altura, assim como os valores antigos.
<code>onDraw(Canvas)</code>	Método responsável por desenhar o componente na tela. Pode ser implementado manualmente para controlar o que é desenhado na tela.

Atributo	Descrição (cont.)
<code>onKeyDown(int keyCode, KeyEvent event)</code> e <code>onKeyUp(int keyCode, KeyEvent event)</code>	Chamados quando o usuário pressiona uma tecla no caso de dispositivos com teclado físico. Você pode até sobrescrever esses métodos na sua classe de Activity para recuperar as teclas digitadas pelo usuário.
<code>onTouchEvent(MotionEvent)</code>	Chamado quando o usuário toca na view.

Nos próximos tópicos, vamos estudar vários tipos de views que podemos utilizar para criar os layouts de tela no Android.

7.6 TextView e EditText – campo de texto para digitar informações

A primeira e mais simples das subclasses de View é o `android.widget.TextView`, que representa um texto/label. E a classe `android.widget.EditText` é uma subclasse de `TextView` utilizada para criar um campo de texto.

O `EditText` pode ser usado para entrada de texto normal, ou para aceitar apenas números ou campos de senha. A seguir, podemos visualizar um exemplo de um formulário de login, em que é demonstrado como criar um campo de texto normal para o login e um campo de senha.



/res/layout/exemplo_edittext.xml

```
<?xml version="1.0" encoding="utf-8"?>
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:stretchColumns="1" android:padding="16dp">
    <TableRow>
        <TextView android:text="@string/usuario" android:padding="3dip" />
        <EditText android:id="@+id/campoLogin" android:padding="3dip" android:inputType="text"/>
    </TableRow>
    <TableRow>
        <TextView android:text="@string/senha" />
        <EditText android:id="@+id/campoSenha" android:inputType="textPassword" />
    </TableRow>
    <TableRow android:gravity="right">
        <Button android:id="@+id/login" android:text="@string/login" />
    </TableRow>
</TableLayout>
```

O `EditText` precisa definir o tipo de entrada do texto. Isso é feito pelo atributo `android:inputType`. O padrão para texto normal é `android:inputType="text"`, mas você pode configurar para o modo de entrada de senha com o atributo `android:inputType="textPassword"`, ou `android:inputType="number"` para números, dentre outras configurações.

7.7 AutoCompleteTextView

A classe `android.widget.AutoCompleteTextView` é um campo de texto que completa automaticamente o texto que o usuário está digitando, e é muito útil em determinados casos. Existem dois parâmetros que podem ser informados no momento de criar essa classe.

- `android:completionThreshold` – Número de letras que o usuário precisa digitar para iniciar o autopreenchimento do texto.
- `android:completionHint` – Texto utilizado para exibir uma dica sobre o preenchimento do texto. O texto é exibido na parte inferior do popup com as opções quando este é aberto.

Podemos visualizar a seguir um exemplo que cria um campo de texto que vai sugerir os nomes dos estados do Brasil. Para preencher a lista, é utilizada a classe `android.widget.ArrayAdapter`, que é uma implementação da interface `android.widget.ListAdapter`. Um adapter é utilizado para fazer a ligação entre o conteúdo e o componente, ou seja, ele fornece o conteúdo para preencher determinado componente. Vamos estudar o conceito de adapters mais para frente; portanto, fique tranquilo.



ExemploAutoCompleteTextViewActivity.java

```
public class ExemploAutoCompleteTextViewActivity extends Activity {
    private static final String[] ESTADOS = new String[] { "Acre", "Alagoas", "Amapá", "Amazonas",
        "Bahia", "Ceará", "...", "São Paulo", "Santa Catarina", "Sergipe", "Tocantins" };
    @Override
    protected void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_exemplo_auto_complete_textview);
        AutoCompleteTextView estados = (AutoCompleteTextView) findViewById(R.id.estados);
        // Adapter para preencher com os estados
        ArrayAdapter<String> adaptador = new ArrayAdapter<String>(this,
            android.R.layout.simple_dropdown_item_1line, ESTADOS);
        estados.setAdapter(adaptador);
    }
}
```

res/layout/activity_exemplo_auto_complete_textview.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="wrap_content"
    android:orientation="vertical" android:padding="10dp">
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Estados" />
    <AutoCompleteTextView android:id="@+id/estados"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:completionThreshold="1"
        android:completionHint="Digite o nome de um estado" />
    <Button
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="OK" />
</LinearLayout>
```

Nesse exemplo, o atributo `android:completionThreshold` foi definido com o valor 1, indicando que o popup com o autopreenchimento deve ser aberto quando o usuário digitar a primeira letra. O atributo `android:completionHint` define a mensagem que aparece na parte inferior do popup. A figura 7.2 exibe o resultado. Como foi digitado o texto "Par", os estados Para, Paraíba e Paraná foram sugeridos.

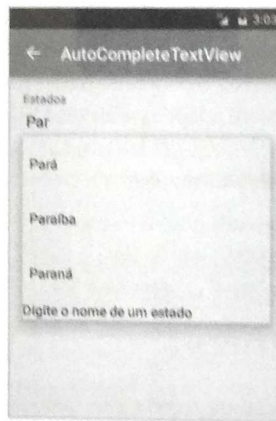


Figura 7.2 – Exemplo da classe `AutoCompleteTextView`.

Nota: para melhor acompanhamento da leitura, execute o projeto de exemplo deste capítulo no emulador. A activity inicial contém uma listagem com todos os exemplos.

7.8 Button e ImageButton

As classes `android.widget.Button` e `android.widget.ImageButton` são utilizadas para criar um botão na tela. A diferença é que a classe `ImageButton` permite usar uma imagem para desenhar o botão. O exemplo a seguir utiliza a classe `ImageButton`, que exibe um alerta quando o botão é clicado.



ExemploImageButtonActivity.java

```
public class ExemploImageButtonActivity extends Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_exemplo_image_button);
        ImageButton botaoImagem1 = (ImageButton) findViewById(R.id.img1);
        final Context context = this;
        botaoImagem1.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                Toast.makeText(context, "Imagem 1 OK", Toast.LENGTH_SHORT).show();
            }
        });
        ImageButton botaoImagem2 = (ImageButton) findViewById(R.id.img2);
        botaoImagem2.setImageResource(R.drawable.smile2);
        botaoImagem2.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
                Toast.makeText(context, "Imagem 2 OK", Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```



/res/layout/activity_exemplo_imagem_button.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical">
    <ImageButton android:id="@+id/img1" android:layout_width="match_parent"
        android:layout_height="wrap_content" android:src="@drawable/smile1" />
    <ImageButton android:id="@+id/img2" android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</LinearLayout>
```


Observe que a primeira imagem é definida diretamente no arquivo de layout com o atributo `android:src="@drawable/smile1"`. Entretanto, a segunda imagem é definida dinamicamente no código.

```
botaoImagem2.setImageResource(R.drawable.smile2);
```

Os métodos que podem ser utilizados para alterar a imagem dinamicamente podem ser visualizados na lista a seguir. Esses métodos estão definidos na classe `ImageView`, a superclasse de `ImageButton`.

Método	Descrição
<code>setImageBitmap(bitmap)</code>	Recebe um <code>android.graphics.Bitmap</code> para exibir a imagem.
<code>setImageResource(id)</code>	Recebe o id da imagem utilizando a constante <code>R.drawable.imagem</code> .
<code>setImageURI(uri)</code>	Recebe uma URI para exibir a imagem. Uma URI é criada com o método <code>Uri.parse(string)</code> e pode representar o caminho para um arquivo ou um link para uma imagem na web.

A figura 73 demonstra o resultado deste exemplo executando no emulador. Observe que a primeira imagem ocupou a largura da tela inteira, porque sua largura foi definida como `match_parent`.

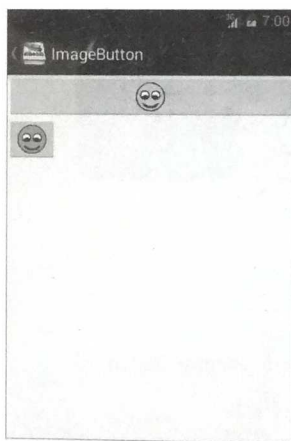


Figura 73 – Exemplo de `ImageButton`.

Outro conceito importante sobre botões são os seletores de estado (selectors), utilizados para definir uma imagem diferente dependendo do estado do botão, como, por exemplo, se o botão está clicado ou não. Para isso, você deve inserir na pasta `/res/drawable` um arquivo XML com a tag `<selector>` com cada estado da imagem. Esse arquivo agrupa um conjunto de imagens, e será transformado em uma imagem pelo compilador.

`/res/drawable/exemplo_seletores.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android">
<!-- pressed -->
    <item android:state_pressed="true" android:drawable="@drawable/button_pressed" />
<!-- focused -->
    <item android:state_focused="true" android:drawable="@drawable/button_focused" />
<!-- hovered -->
    <item android:state_hovered="true" android:drawable="@drawable/button_focused" />
    <item android:drawable="@drawable/button_normal" /> <!-- default -->
</selector>
```

O arquivo XML define uma imagem para cada tipo de estado do botão. Portanto, você precisa pedir aos designers para criar essas imagens, ou pelo menos duas: normal e selecionado. Com esse arquivo de seletores (selectors) em mãos, basta utilizá-lo como a imagem de fundo para o botão. Na prática, o arquivo XML vai virar uma imagem e você pode utilizar as notações `@drawable` e `R.drawable` para acessar esse recurso.

```
<Button android:layout_height="wrap_content" android:layout_width="wrap_content"
    android:background="@drawable/exemplo_seletores" />
```

7.9 CheckBox e ToggleButton

Um checkbox pode ser criado no Android com a classe `android.widget.CheckBox`, que pode ser inserida facilmente em um arquivo de layout conforme o exemplo a seguir.

```
<CheckBox android:id="@+id/checkReceberEmail"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:text="Receber email" />
```

Para verificar se o checkbox está marcado ou não, utilize o método `isChecked()`.

```
CheckBox check = (CheckBox) findViewById(R.id.checkReceberEmail);
boolean receberEmail = check.isChecked();
```

Outra classe que pode ser utilizada para selecionar uma opção, similar ao checkbox, é a `android.widget.ToggleButton`.

```
<ToggleButton android:id="@+id/toggle"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:textOn="Ligado" android:textOff="Desligado" />
```

Essa classe também contém o método `isChecked()`. Com os atributos `android:textOn` e `android:textOff`, é possível controlar os textos que aparecem no botão quando ele está selecionado ou não. A seguir, podemos visualizar um exemplo das classes `CheckBox` e `ToggleButton`.

 `/res/layout/activity_exemplo_toggle_button.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical">
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Exemplo de CheckBox e ToggleButton" />
    <CheckBox android:id="@+id/check1"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Check 1" />
    <CheckBox android:id="@+id/check2"
        style="?android:attr/starStyle"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Check 2" />
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="O ToggleButton mostra os textos Ligado ou Desligado..." />
    <ToggleButton android:id="@+id/toggle"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:textOn="Ligado" android:textOff="Desligado" />
    <Button android:id="@+id/btOK"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="OK" />
</LinearLayout>
```

 `ExemploToggleButtonActivity.java`

```
public class ExemploToggleButtonActivity extends Activity {
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_exemplo_toggle_button);
        final ToggleButton toggle = (ToggleButton) findViewById(R.id.toggle);
        Button b = (Button) findViewById(R.id.btOK);
        b.setOnClickListener(new View.OnClickListener() {
            public void onClick(View v) {
```

```

        boolean selecionado = toggle.isChecked();
        Toast.makeText(ExemploToggleButton.this, "Selecionado: " + selecionado,
            Toast.LENGTH_SHORT).show();
    }
}
}
}

```

A figura 7.4 demonstra o resultado desse exemplo. Observe que o segundo checkbox está parecido com uma estrela porque foi definido um estilo customizado com o atributo `style="?android:attr/starStyle"`.

Esse é um exemplo legal para explicar algo importante do Material Design: como as cores são aplicadas no tema. Se você executar o projeto de exemplo no emulador, verá que a cor da action bar é azul e a cor dos componentes é vermelha. Por exemplo, ao marcar o checkbox, a cor da seleção fica vermelha. Isso é configurado de forma simples no arquivo `/res/values-v21/styles.xml`, que sobrescreve algumas propriedades de cores. Veja que o projeto `Demo-Views` não utiliza a biblioteca de compatibilidade, e por isso o tema Material foi configurado apenas para API Level 21 ou superior.

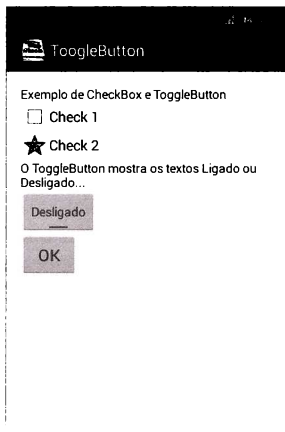


Figura 7.4 – Exemplo de CheckBox e ToggleButton.

 `/res/values-v21/styles.xml`

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="AppTheme" parent="android:Theme.Material.Light">
        <!-- Cor primária -->
        <item name="android:colorPrimary">@color/primary</item>
    </style>
</resources>

```

```

<!-- Variação escura da cor primária para a status bar e contextual app bars -->
<item name="android:colorPrimaryDark">@color/primary_dark</item>
<!-- Cor de acentuação para as views (checkbox, textfield etc.) -->
<item name="android:colorAccent">@color/accent</item>
</style>
</resources>

```

Para entender o significado de cada cor, execute o projeto em um emulador com Android 5.0 ou superior e veja os comentários no código. As cores estão definidas no arquivo `/res/values/colors.xml`.



`/res/values/colors.xml`

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- Cores do Material Theme -->
    <color name="primary">#03A9F4</color>    <!-- Azul -->
    <color name="primary_dark">#01579B</color> <!-- Azul escuro -->
    <color name="accent">#F44336</color>    <!-- Vermelho -->
    <!-- Cores -->
    <color name="vermelho">#ff0000</color>
    <color name="azul">#0000ff</color>
    <color name="verde">#00ff00</color>
    <color name="branco">#ffffff</color>
</resources>

```

Nota: no capítulo 11, sobre Material Design, vamos voltar a estudar essas cores. De qualquer forma, é importante você entender que a cor primária (primary) representa a cor da marca do cliente ou do seu aplicativo, e a cor de acentuação (accent) é utilizada para destacar as views e determinados componentes na tela.

7.10 RadioButton

O componente radio button permite selecionar apenas uma única opção de uma lista. No Android, as classes `android.widget.RadioGroup` e `android.widget.RadioButton` são utilizadas para isso. A classe `RadioGroup` define o grupo que contém a lista de opções, na qual cada opção é representada por um `RadioButton`.

O exemplo a seguir cria dois botões com os textos **Sim** e **Não**, respectivamente:

```

<RadioGroup android:layout_width="match_parent"
    android:layout_height="wrap_content" android:orientation="horizontal"

```

```

        android:id="@+id/group1">
        <RadioButton android:id="@+id/radioSim"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="Sim" android:checked="false" />
        <RadioButton android:id="@+id/radioNao"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="Não" android:checked="false" />
    </RadioGroup>

```

Observe que cada `RadioButton` tem um `id`, para que posteriormente o `id` do botão selecionado possa ser recuperado, chamando o método `getCheckedRadioButtonId()` da classe `RadioGroup`. Nesse exemplo, foram definidos os `ids` `radioSim` e `radioNao`. Isso possibilita utilizar o seguinte código para descobrir qual botão foi selecionado:

```
boolean sim = R.id.radioSim == group.getCheckedRadioButtonId();
```

Dessa forma, o `id` do `RadioButton` selecionado está sendo comparado com um `id` conhecido que foi definido no arquivo XML de layout. Para demonstrar o uso do `RadioButton` e do `CheckBox`, criaremos um exemplo de um formulário.

 /res/layout/activity_exemplo_check_radio_form.xml

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical">
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Nome" />
    <EditText android:id="@+id/textNome"
        android:layout_width="match_parent" android:layout_height="wrap_content" />
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Concorda?" />
    <RadioGroup android:layout_width="match_parent"
        android:layout_height="wrap_content" android:orientation="horizontal"
        android:id="@+id/group1">
        <RadioButton android:id="@+id/radioSim"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="Sim"
            android:checked="false" />
        <RadioButton android:id="@+id/radioNao"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="Não"
            android:checked="false"/>
    </RadioGroup>

```

```

</RadioGroup>
<TextView
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:text="Receber Email ?" />
<CheckBox android:id="@+id/checkReceberEmail"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:text="Receber email" />
<Button android:id="@+id/buttonEnviar"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:text="Enviar" />
</LinearLayout>

```

O seguinte código mostra como verificar se o checkbox e radio button estão selecionados:



ExemploCheckRadioFormActivity.java

```

public class ExemploCheckRadioFormActivity extends Activity {
    private static final String TAG = "livro";
    @Override
    protected void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_exemplo_check_radio_form);
        final EditText textNome = (EditText) findViewById(R.id.textNome);
        final RadioGroup group = (RadioGroup) findViewById(R.id.group1);
        group.setOnCheckedChangeListener(new RadioGroup.OnCheckedChangeListener() {
            public void onCheckedChanged(RadioGroup group, int checkedId) {
                boolean sim = R.id.radioSim == checkedId;
                boolean nao = R.id.radioNao == checkedId;
                if (sim) {
                    Log.i(TAG, "Marcou radio Sim: " + checkedId);
                } else if (nao) {
                    Log.i(TAG, "Marcou radio Não: " + checkedId);
                }
            }
        });
        final CheckBox check = (CheckBox) findViewById(R.id.checkReceberEmail);
        // Define um listener para executar quando alterar o check
        check.setOnCheckedChangeListener(new CheckBox.OnCheckedChangeListener() {
            public void onCheckedChanged(CompoundButton buttonView, boolean isChecked) {
                Log.i(TAG, "check: " + isChecked);
            }
        });
    }
}

```

```

Button b = (Button) findViewById(R.id.buttonEnviar);
b.setOnClickListener(new OnClickListener() {
    public void onClick(View v) {
        Log.i(TAG, "OK");
        // Compara o id do radioSim
        boolean concorda = R.id.radioSim == group.getCheckedRadioButtonId();
        boolean receberEmail = check.isChecked();
        StringBuffer sb = new StringBuffer();
        sb.append("Nome: ").append(textNome.getText())
            .append("\nReceber Email: ").append(receberEmail ? "Sim" : "Não")
            .append("\nConcorda: ").append(concorda ? "Sim" : "Não");
        Toast.makeText(ExemploCheckRadio.this, sb.toString(), Toast.LENGTH_SHORT).show();
    }
});
}
}
}

```

Observe que ambas as classes `CheckBox` e `RadioGroup` contêm o método `setOnCheckedChangeListener(listener)` para configurar o listener que deve ser executado quando o estado do checkbox ou radio for alterado. Porém, como demonstrado no exemplo, os métodos recebem diferentes interfaces (têm os mesmos nomes, mas estão em pacotes diferentes). Ao executar esse exemplo, podemos visualizar o formulário, conforme mostra a figura 7.5. Ao preencher o formulário e pressionar o botão, um alerta com as opções selecionadas é exibido.

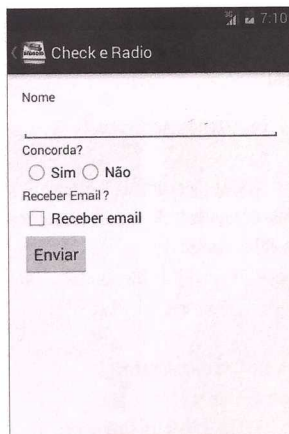


Figura 7.5 – Exemplo de `CheckBox` e `RadioButton`.

7.11 Spinner

A classe `android.widget.Spinner` é utilizada para criar um combo com opções na tela. Para definir a lista que deve ser exibida no combo, é usada uma implementação de `android.widget.SpinnerAdapter` que herda de `android.widget.Adapter`.

A seguir, podemos visualizar um exemplo que cria um combo na tela que lista os nomes dos planetas do sistema solar. Ao selecionar algum planeta, sua foto é exibida na tela.

 `/res/layout/activity_exemplo_spinner.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="wrap_content"
    android:orientation="vertical">
    <TextView
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="Selecione uma opção" />
    <Spinner android:id="@+id/comboPlanetas"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:drawSelectorOnTop="true"
        android:prompt="@string/texto_combo" />
    <ImageView android:id="@+id/img"
        android:layout_width="match_parent" android:layout_height="match_parent" />
</LinearLayout>
```

 `ExemploSpinnerActivity.java`

```
public class ExemploSpinnerActivity extends Activity {
    // Planetas
    private int[] imagens = { R.drawable.mercurio, R.drawable.venus, R.drawable.terra,
        R.drawable.marte, R.drawable.jupiter, R.drawable.saturno, R.drawable.urano,
        R.drawable.netuno, R.drawable.plutao };
    private String[] planetas = new String[] { "Mercúrio", "Vênus", "Terra", "Marte",
        "Júpiter", "Saturno", "Urano", "Netuno", "Plutão"};
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_exemplo_spinner);
        final ImageView imagem = (ImageView) findViewById(R.id.img);
        final Spinner combo = (Spinner) findViewById(R.id.comboPlanetas);
        ArrayAdapter<String> adaptador = new ArrayAdapter<String>(this,
            android.R.layout.simple_spinner_item, planetas);
```

```
adaptador.setDropDownViewResource(android.R.layout.simple_spinner_item);
combo.setAdapter(adaptador);

// Se seleccionar algum planeta atualiza a imagem
combo.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> parent, View v, int posicao, long id) {
        // Atualiza a imagem
        imagem.setImageResource(imagens[posicao]);
    }
    @Override
    public void onNothingSelected(AdapterView<?> parent) { }
});
}
```

Se for necessário obter o item selecionado, utilize algum dos seguintes métodos:

Método	Descrição
<code>Object getItem()</code>	Retorna o item selecionado.
<code>long getItemId()</code>	Retorna o id do item selecionado.
<code>int getItemPosition()</code>	Retorna a posição do item selecionado. Essa posição é equivalente ao array fornecido para o spinner.

Nesse exemplo, preferimos monitorar o estado do `Spinner` em tempo de execução. Para isso, implementamos a interface `AdapterView.OnItemClickListener` e o método `onItemClick(parent,view,posicao,id)`, com o objetivo de atualizar a imagem automaticamente quando um planeta for selecionado. A figura 7.6 mostra o planeta Terra selecionado no combo.

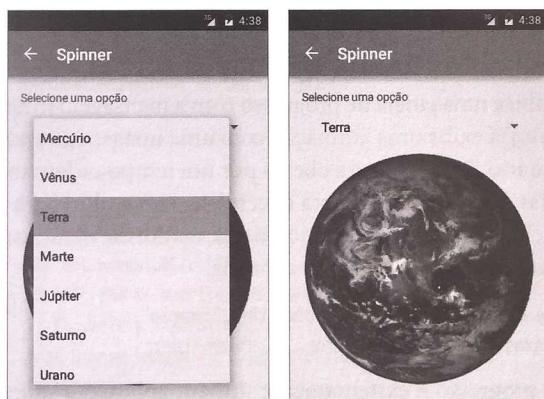


Figura 7.6 – Planeta Terra selecionado.

Observe que foi utilizado o recurso nativo `android.R.layout.simple_spinner_item` para criar o drop-down com a lista de planetas.

```
adaptador.setDropDownViewResource(android.R.layout.simple_spinner_item);
```

Você pode criar seu próprio recurso XML se desejar customizar a interface, ou até utilizar outro padrão da plataforma como o recurso `android.R.layout.simple_spinner_dropdown_item`. Altere o exemplo para esse código e confira o resultado.

```
adaptador.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
```

Nota: a classe `android.R` acessa os recursos nativos do Android. Cada projeto contém sua própria classe `R` com o seu pacote específico; portanto, sempre preste atenção ao fazer os imports.

7.12 ProgressDialog – janela de progresso

Você já deve ter visto aquela mensagem de “Por favor, aguarde...” em alguma aplicação que faz algum processamento demorado. No Android, existe uma classe especial justamente para exibir uma janela na tela com uma mensagem para aguardar. Para isso, usaremos a classe `android.app.ProgressDialog`, filha da classe `android.app.Dialog`.

A seguir, podemos ver um exemplo muito simples de como usar a classe `ProgressDialog`.

```
ProgressDialog dialog = new ProgressDialog(this);  
dialog.setTitle("Exemplo");  
dialog.setMessage("Buscando imagem, por aguarde...");  
dialog.setIndeterminate(true); // Indica para executar por tempo indeterminado  
dialog.setCancelable(true);   // Se pode cancelar caso pressione o voltar  
dialog.show();
```

Esse código exibirá uma janela de progresso com a mensagem informada. O próprio `ProgressDialog` já exibe uma animação com uma imagem girando para encher os olhos do usuário. A janela ficará aberta por um tempo indeterminado até que o método `dismiss()` seja chamado para encerrá-la. Um atalho para essas linhas é simplesmente chamar o método estático `show()`, conforme demonstrado a seguir, que faz a mesma coisa que o exemplo anterior.

```
ProgressDialog dialog = ProgressDialog.show(this, "Exemplo",  
    "Buscando imagem, por favor, aguarde...", false, true);
```

Essa janela de progresso é extremamente útil em aplicações que acessam a internet para buscar informações, nas quais o tempo de resposta é indeterminado.

Demonstraremos um exemplo que faz o download de uma imagem da internet. Enquanto o download está em andamento, um alerta de progresso é exibido na tela. O arquivo de layout a seguir contém apenas a imagem que será atualizada depois que o download terminar.



/res/layout/activity_exemplo_progress_dialog.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent" >
    <ImageView android:id="@+id/img" android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:scaleType="fitCenter" />
</FrameLayout>
```



ExemploProgressDialogActivity.java

```
public class ExemploProgressDialogActivity extends Activity {
    private static final String URL = "http://livroandroid.com.br/ings/livro_android.png";
    private ProgressDialog dialog;
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_exemplo_progress_dialog);
        // Abre a janela com a barra de progresso
        dialog = ProgressDialog.show(this,"Exemplo",
            "Buscando imagem, por favor, aguarde...", false,true);
        downloadImagem(URL);
    }
    // Faz o download da imagem em uma nova thread
    private void downloadImagem(final String urlImg) {
        new Thread() {
            @Override
            public void run() {
                try {
                    // Faz o download da imagem
                    URL url = new URL(urlImg);
                    InputStream in = url.openStream();
                    // Converte a InputStream do Java para Bitmap
                    final Bitmap imagem = BitmapFactory.decodeStream(in);
                    in.close();
                    // Atualiza a tela
                }
            }
        }.start();
    }
}
```

```

        atualizaImagem(imagem);
    } catch (IOException e) {
        // Uma aplicação real deveria tratar este erro
        Log.e("Erro ao fazer o download: ", e.getMessage(), e);
    }
}
}.start();
}

private void atualizaImagem(final Bitmap imagem) {
    runOnUiThread(new Runnable() { // Este código é necessário, pois foi aberta uma thread
        @Override
        public void run() {
            // Fecha a janela de progresso
            dialog.dismiss();
            ImageView imgView = (ImageView) findViewById(R.id.img);
            imgView.setImageBitmap(imagem);
        }
    });
}
}
}

```

Para o exemplo funcionar, declare a permissão `INTERNET` no arquivo *AndroidManifest.xml*.



AndroidManifest.xml

```

<manifest ... />
    <uses-permission android:name="android.permission.INTERNET" />
    <application ... />
</manifest>

```

A figura 7.7 mostra a janela de progresso aberta enquanto a aplicação está fazendo o download da imagem em segundo plano. Depois que o download é concluído, o método `dismiss()` da classe `ProgressDialog` é chamado para fechar a janela e a imagem é exibida.

A parte do código que faz o `ProgressDialog` é teoricamente simples, mas esse exemplo mostrou dois problemas que você vai encontrar ao criar qualquer funcionalidade que acesse a internet.

1. É obrigatório iniciar uma thread para fazer qualquer operação de I/O, como acessar a internet, ler arquivos e consultar banco de dados, ou qualquer operação demorada. Se a thread não for criada, dependendo do caso, o Android pode lançar uma exceção ou a aplicação pode mostrar o erro ANR (Android Not Responding). Para mais detalhes, consulte o capítulo 10, sobre threads.

2. No Android, cada aplicação executa em um único processo, e cada processo contém uma thread dedicada. Essa thread também é responsável por desenhar e tratar todos os eventos da interface gráfica, conhecida popularmente como UI Thread ou Main Thread. Existe uma regra no Android que diz: somente a UI Thread pode atualizar a interface, ou seja, somente ela pode chamar qualquer método que vai atualizar uma view. Por isso, neste exemplo foi utilizado o método `runOnUiThread(runnable)` para sincronizar o código que vai atualizar a view com a UI Thread. Na prática o método `runOnUiThread(runnable)` faz com que o `Runnable` passado como parâmetro execute na UI Thread. Se você não fizer isso a aplicação vai lançar uma exceção. Para mais detalhes sobre esse assunto, consulte o capítulo 10, sobre a classe `Handler`.

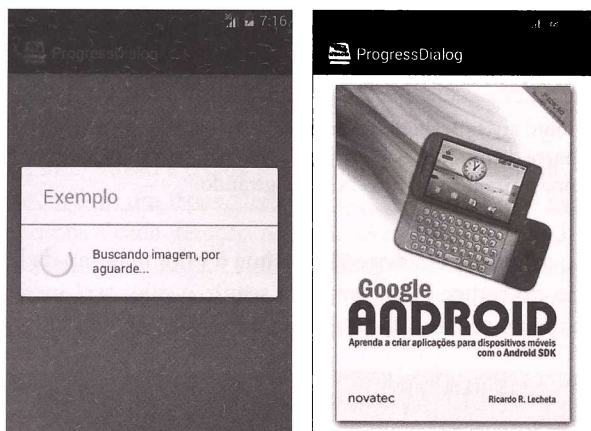


Figura 7.7 – Exemplo de ProgressDialog.

7.13 ProgressBar – barra de progresso

A classe `android.widget.ProgressBar` é utilizada para exibir uma barra de progresso na tela, que pode durar por tempo indeterminado ou não. Com uma barra de progresso, é possível incrementar o valor da barra, para ela ser preenchida aos poucos, à medida que o processamento vai terminando. Dessa forma, é possível dar ao usuário uma ideia de quando determinada tarefa será concluída.

A seguir, veremos um exemplo que utiliza a classe `ProgressBar`.


```

        final int progress = i;
        // Atualiza a barra de progresso
        runOnUiThread(new Runnable() {
            public void run() {
                Log.d(TAG, ">> Progress: " + progress);
                mProgress.setProgress(progress);
            }
        });
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {}
    }
    Log.d(TAG, "Fim.");
}
}).start();
});
}

```

Nesse código, uma thread é criada para simular o processamento, e para demorar um pouco foi feito um `Thread.sleep(200)`, que coloca a thread para dormir por 200 milissegundos a cada iteração. A figura 7.8 mostra o resultado com a barra de progresso sendo atualizada. Lembre-se de que, se você executar o projeto de exemplo deste capítulo no emulador, a barra de progresso será vermelha, que foi a cor definida para "accent color" do tema Material.

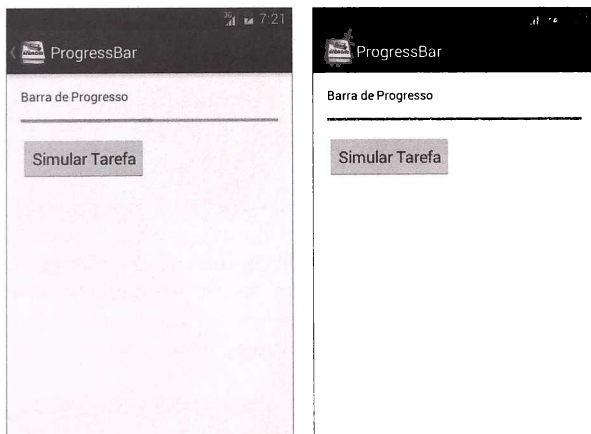


Figura 7.8 – Exemplo de ProgressBar.

7.14 Toast – alertas rápidos

A classe `android.widget.Toast` é utilizada para mostrar alertas para o usuário. Cada alerta pode ser visualizado na tela por um tempo curto, especificado pela constante `Toast.LENGTH_SHORT`, ou por um tempo longo se utilizar a constante `Toast.LENGTH_LONG`. A forma mais simples de criar um alerta é com o método `Toast.makeText(contexto, mensagem, tempo)`:

```
Toast toast = Toast.makeText(this, "Teste de Mensagem", Toast.LENGTH_SHORT);
toast.show();
```

Na maioria dos casos, somente o método anterior já é suficiente para exibir mensagens de alerta para o usuário. Entretanto, pode ser que seja necessário exibir um alerta com uma interface mais complexa. Para isso, a classe `Toast` contém o método `setView(view)`, o qual pode ser chamado para configurar a `View` que será exibida no alerta, que pode ser simplesmente uma imagem definida pela classe `ImageView` ou mesmo uma tela com um layout complexo.

Por exemplo, para criar um alerta que no lugar de algum texto exiba uma imagem, poderíamos utilizar o seguinte trecho de código:

```
ImageView imagem = new ImageView(this);
imagem.setImageResource(R.drawable.smile);
Toast toast = new Toast(this);
toast.setView(imagem);
toast.setDuration(Toast.LENGTH_LONG);
toast.show();
```

Observe que o método `setView(view)` pode receber qualquer subclasse de `View`, inclusive um gerenciador de layout, o que permite exibir um alerta com uma interface bem customizada.

Nota: um toast é aquele alerta rápido muito conhecido por desenvolvedores e usuários do Android. Esse alerta não tem vínculo com o que foi ou está sendo executado, de forma que o usuário pode continuar fazendo o que quiser no celular. Por exemplo, no aplicativo do Gmail, logo depois de enviar um email, um toast mostra uma mensagem de que o email está sendo enviado. Mas você pode continuar usando o celular normalmente, pois o toast é apenas uma mensagem de informação sem estado.

7.15 AlertDialog – alertas para o usuário confirmar

A classe `Toast` que estudamos anteriormente mostra uma mensagem temporária com a qual o usuário não pode interagir. Se for necessário que o usuário pressione no botão **OK** do alerta, ou responda uma pergunta do tipo **Sim** ou **Não**, podemos utilizar a classe `AlertDialog`.

```
Builder builder = new AlertDialog.Builder(this);
builder.setIcon(R.drawable.ic_launcher);
builder.setTitle("Título");
builder.setMessage("Mensagem");
builder.setPositiveButton("Sim", new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int which) {
        Toast.makeText(getBaseContext(), "Clicou em Sim!", Toast.LENGTH_SHORT).show();
        return;
    }
});
builder.setNegativeButton("Não", new DialogInterface.OnClickListener() {
    public void onClick(DialogInterface dialog, int Não) {
        Toast.makeText(getBaseContext(), "Clicou em Sim!", Toast.LENGTH_SHORT).show();
        return;
    }
});
AlertDialog dialog = builder.create();
dialog.show();
```

A figura 7.9 mostra o resultado do alerta desse código. O alerta obriga o usuário a pressionar o botão **Sim** ou **Não**, o que é diferente do `Toast`, que exibe temporariamente a mensagem e desaparece.

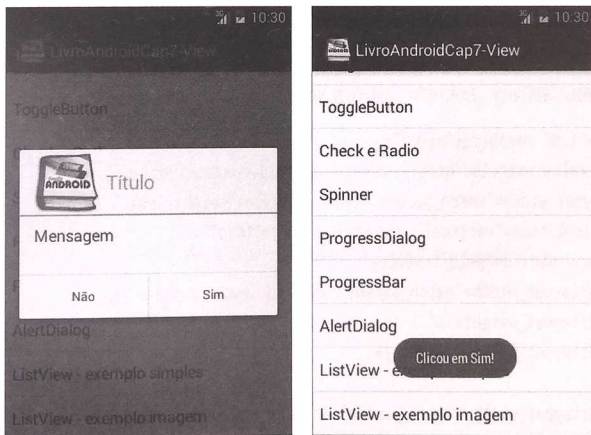


Figura 7.9 – Alerta com botões de sim e não.

7.16 LayoutInflater – inflando um arquivo XML

A classe `android.view.LayoutInflater` é utilizada para converter um arquivo XML para um objeto do tipo `view`, o que é conhecido pelos desenvolvedores como “inflar um layout”. O `LayoutInflater` é um serviço do sistema e deve ser recuperado com o seguinte código:

```
LayoutInflater inflate = (LayoutInflater)
    context.getSystemService(Context.LAYOUT_INFLATER_SERVICE);
```

Nesse caso, o `context` é uma referência para a classe `Activity`, ou muitas vezes é o `this`. Também podemos utilizar o seguinte atalho para obter o `LayoutInflater`.

```
LayoutInflater inflate = LayoutInflater.from(context);
```

Depois de obter o `LayoutInflater`, basta chamar o método `inflate(id,parent)`, o qual recebe o `id` do arquivo XML desejado e retorna uma instância de um objeto do tipo `View`. O seguinte código mostra como inflar o arquivo `/res/layout/inflate_teste.xml` e retornar um objeto do tipo `View` que pode ser utilizado no código. Isso permite criar uma `view` como XML e inflar o objeto, em vez de utilizar diretamente a API.

```
View view = (View) LayoutInflater.from(this).inflate(R.layout.inflate_teste, layout, false);
```

O caso de uso mais comum do `LayoutInflater` é para criar as `views` de um `adapter`; por exemplo, para preencher os componentes `ListView` e `ViewPager`.

7.17 ListView

A classe `android.widget.ListView` é um dos componentes visuais mais utilizados e representa uma lista na tela. O exemplo a seguir mostra um arquivo de layout com um `ListView`.

 `/res/layout/activity_exemplo_listview.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" android:padding="16dp" >
    <ListView android:id="@+id/listview"
        android:layout_width="match_parent" android:layout_height="0dp"
        android:layout_weight="1"
        android:layout_margin="10dp" />
    <View
        android:layout_width="match_parent" android:layout_height="20dp"
        android:background="@color/primary" />
</LinearLayout>
```

Quem fornece os dados para preencher o `ListView` é um adapter, que é uma classe que implementa a interface `android.widget.ListAdapter`. Opcionalmente podemos estender a classe `android.widget.BaseAdapter` que já implementa essa interface e deixa poucos métodos abstratos para terminarmos a implementação. O código a seguir mostra um adapter que vai preencher o `ListView` com uma lista que tem os nomes dos planetas. Para cada planeta será criada uma view, que nesse exemplo é um simples `TextView`. Veja os comentários no código para entender o que faz cada método do adapter.



SimplesAdapter.java

```
public class SmplesAdapter extends BaseAdapter {
    private String[] planetas = new String[] { "Mercúrio", "Vênus", "Terra", "Marte",
        "Júpiter", "Saturno", "Urano", "Netuno", "Plutão"};
    private Context context;
    public SmplesAdapter(Context context) {
        super();
        this.context = context; // O context é necessário para criar a view
    }
    @Override
    public int getCount() {
        return planetas.length; // Retorna a quantidade de itens do adapter
    }
    @Override
    public Object getItem(int position) {
        return planetas[position]; // Retorna o objeto para esta posição
    }
    @Override
    public long getItemId(int position) {
        return position; // Retorna o id do objeto para esta posição
    }
    @Override
    // Retorna a view para esta posição
    public View getView(int position, View convertView, ViewGroup parent) {
        String planeta = planetas[position];
        TextView t = new TextView(context);
        float dip = 50;
        float densidade = context.getResources().getDisplayMetrics().density; // Densidade
                                                // da tela

        int px = (int) (dip * densidade + 0.5f);
        t.setHeight(px);
        t.setText(planeta);
        return t;
    }
}
```

O código do adapter está criando um `TextView` por programação e definindo a altura como 50dp. Veja no código que o número 50 não é utilizado diretamente para definir a altura do `TextView`, pois isso traria resultados diferentes conforme a resolução e densidade da tela do dispositivo. Por isso, esse cálculo converteu 50dp para 50px. Mas nem sempre 50dp é 50px, pois o resultado da conversão pode ser 100px, 150px, 200px etc., conforme a densidade da tela. Por isso a conversão é necessária. Mas não se preocupe com isso agora, estou apenas alertando sobre um problema comum. Nós vamos estudar esse assunto em mais detalhes no capítulo 30, sobre como criar aplicativos compatíveis com diferentes tamanhos de telas.

Para finalizar o exemplo, segue o código-fonte da activity. Para preencher a lista, é preciso chamar o método `setAdapter(adapter)` do `ListView` informando o adapter. A lista terá a quantidade de linhas que o adapter retornar no método `getCount()`.



ExemploListViewActivity.java

```
public class ExemploListViewActivity extends Activity implements OnItemClickListener {
    protected static final String TAG = "livro";
    private ListView listView;
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_exemplo_listview);
        // ListView
        listView = (ListView) findViewById(R.id.listview);
        listView.setAdapter(new SimpleAdapter(this));
        listView.setOnItemClickListener(this);
    }
    public void onItemClick(AdapterView<?> parent, View view, int idx, long id) {
        // Objeto selecionado, que neste caso era de um array de strings
        String s = (String) parent.getAdapter().getItem(idx);
        Toast.makeText(this, "Texto selecionado: " + s + ", posição: " + idx,
            Toast.LENGTH_SHORT).show();
    }
}
```

Ao clicar em algum item da lista, o método `onItemClick()` será chamado, informando a posição e o id do objeto selecionado. Com base nisso, podemos recuperar o objeto selecionado no código. Ao executar esse exemplo, você verá o `ListView` no centro e na parte interior da tela uma view quadrada que preenchi com o fundo azul (mesma cor da action bar), conforme a figura 7.10. No código do arquivo de layout, veja que a altura do `ListView` foi definida como 0dp, para o `ListView` respeitar

o peso que lhe foi atribuído e esticar até onde ele pode, respeitando essa view que está localizada na parte inferior do layout. Se o `ListView` estivesse com a altura como `match_parent`, ele iria esticar sem respeitar os outros componentes, e a view de baixo seria jogada para fora da tela.

Parabéns! Você acabou de fazer o primeiro exemplo de `ListView`, um dos componentes mais utilizados em aplicativos.



Figura 7.10 – `ListView`.

No código desse último exemplo, o método `getView()` da classe `SimplesAdapter` que fizemos está criando o `TextView` pela API, o que não é o recomendado. O ideal é sempre criar uma view utilizando um arquivo XML e “inflar” esse layout com a classe `LayoutInflater`. Vamos então criar o seguinte layout para o adapter:

```
 /res/layout/adapter_simples.xml

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="wrap_content" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content" android:layout_height="50dp" />
</LinearLayout>
```

Feito isso, altere o código do método `getView()` para inflar o layout. Depois de fazer isso, o resultado será o mesmo de antes, mas desse jeito o código fica bem mais organizado. Nem sempre o layout do adapter será tão simples; o recomendado é sempre separar a interface em arquivos XML.



SimplesAdapter.java

```
public class SimplesAdapter extends BaseAdapter {
    ...
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        String planeta = planetas[position];
        View view = LayoutInflater.from(context).inflate(R.layout.adapter_simples, parent, false);
        TextView t = (TextView) view.findViewById(R.id.text);
        t.setText(planeta);
        return view;
    }
}
```

Note que no código XML do arquivo */res/layout/adapter_simples.xml* foi definida a altura do `TextView` como 50dp, que é a notação da medida de densidade do Android. Dessa forma, conforme a resolução da tela do dispositivo, 50dp será convertido para 50px, 75px, 100px etc. Foi isso que fizemos no código quando utilizamos a API, mas aqui no XML basta utilizar a notação dp e tudo é feito automaticamente.

7.18 ListView com adapter customizado

No próximo exemplo, vamos criar uma lista de objetos e exibi-la no `ListView` com um adapter customizado. A classe `android.widget.ListView` é um dos componentes visuais mais utilizados e representa uma lista na tela. Entender esse exemplo é vital para seu futuro como desenvolvedor Android, até porque demonstra a utilização de adapters.

O exemplo a seguir mostra um arquivo de layout com um `ListView`:



/res/layout/activity_exemplo_listview.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" android:padding="16dp" >
    <ImageView android:src="@drawable/ic_launcher"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <ListView android:id="@+id/listview"
        android:layout_width="match_parent" android:layout_height="0dp"
        android:layout_weight="1"
        android:layout_margin="10dp" />
    <ImageView android:src="@drawable/ic_launcher"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
</LinearLayout>
```

Para preencher a lista do `ListView`, vamos criar uma lista de planetas, então crie a classe `Planeta`. Para o exemplo funcionar, copie as figuras de cada planeta e insira na pasta `/res/drawable`. Elas podem ser encontradas no projeto de exemplo deste capítulo.



Planeta.java

```
public class Planeta {
    public String nome;
    public int img; // R.drawable.xxx
    public Planeta(String nome, int img) {
        this.nome = nome;
        this.img = img;
    }
    public static List<Planeta> getPlanetas() {
        List<Planeta> planetas = new ArrayList<Planeta>();
        planetas.add(new Planeta("Mercúrio", R.drawable.planeta_01_mercurio));
        planetas.add(new Planeta("Vênus", R.drawable.planeta_02_venus));
        planetas.add(new Planeta("Terra", R.drawable.planeta_03_terra));
        planetas.add(new Planeta("Marte", R.drawable.planeta_04_marte));
        planetas.add(new Planeta("Júpiter", R.drawable.planeta_05_jupiter));
        planetas.add(new Planeta("Saturno", R.drawable.planeta_06_saturno));
        planetas.add(new Planeta("Urano", R.drawable.planeta_07_urano));
        planetas.add(new Planeta("Netuno", R.drawable.planeta_08_neptuno));
        planetas.add(new Planeta("Plutão", R.drawable.planeta_09_plutao));
        return planetas;
    }
}
```

A classe `Planeta` permite criar a lista de planetas. No código-fonte da `activity`, basta criar essa lista e configurar o `adapter` no `ListView`.



ExemploListViewActivity.java

```
public class ExemploListViewActivity extends Activity implements OnItemClickListener {
    protected static final String TAG = "livro";
    private ListView listView;
    private List<Planeta> planetas;
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_exemplo_listview);
        // ListView
        listView = (ListView) findViewById(R.id.listview);
        planetas = Planeta.getPlanetas();
    }
}
```



```

        listView.setAdapter(new PlanetaAdapter(this, planetas));
        listView.setOnItemClickListener(this);
    }
    public void onItemClick(AdapterView<?> parent, View view, int idx, long id) {
        Planeta p = this.planetas.get(idx);
        Toast.makeText(this, "Planeta: " + p.nome, Toast.LENGTH_SHORT).show();
    }
}

```

Para o código compilar, precisamos criar a classe de adapter.



PlanetaAdapter.java

```

public class PlanetaAdapter extends BaseAdapter {
    private final Context context;
    private final List<Planeta> planetas;
    public PlanetaAdapter(Context context, List<Planeta> planetas) {
        this.context = context;
        this.planetas = planetas;
    }
    @Override
    public int getCount() {
        return planetas != null ? planetas.size() : 0;
    }
    @Override
    public Object getItem(int position) {
        return planetas.get(position);
    }
    @Override
    public long getItemId(int position) {
        return position;
    }
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        // Infla a view
        View view = LayoutInflater.from(context).inflate(R.layout.adapter_planeta, parent, false);
        // Faz findViewById das views que precisa atualizar
        TextView t = (TextView) view.findViewById(R.id.tNomePlaneta);
        ImageView img = (ImageView) view.findViewById(R.id.imgPlaneta);
        // Atualiza os valores das views
        Planeta planeta = planetas.get(position);
        t.setText(planeta.nome);
        img.setImageResource(planeta.img);
    }
}

```

```

        // Retorna a view deste planeta
        return view;
    }
}

```

O arquivo de layout do adapter pode ser visualizado a seguir:

```

 /res/layout/adapter_planeta.xml

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="?android:attr/listPreferredItemHeight"
    android:gravity="center_vertical"
    android:orientation="horizontal">
    <!-- Foto do planeta -->
    <ImageView android:id="@+id/imgPlaneta"
        android:layout_width="0dp" android:layout_height="wrap_content"
        android:layout_weight="3"
        android:src="@drawable/planeta_03_terra" />
    <!-- Nome do planeta -->
    <TextView android:id="@+id/tNomePlaneta"
        android:layout_width="0dp" android:layout_height="wrap_content"
        android:layout_weight="7"
        android:layout_marginLeft="10dp" android:textColor="#000000" />
</LinearLayout>

```

Dica: no código do layout XML, a altura da view do adapter foi definida como `android:layout_height="?android:attr/listPreferredItemHeight"`. Isso acessa um atributo de dimensão nativo do Android, que retorna a altura recomendada pela plataforma para uma linha do `ListView`.

A figura 7.11 mostra o resultado desse exemplo com a lista de planetas.

Nota: existe uma classe especial de activity que é a `ListActivity`, a qual já declara seu próprio layout com um único `ListView`. Mas eu prefiro sempre estender minhas classes diretamente de activity e adicionar um `ListView` no layout, pois assim o layout fica mais flexível e você tem controle do que está fazendo. Para sua consulta, a `MainActivity` do projeto de exemplo deste capítulo é filha de `ListActivity`.

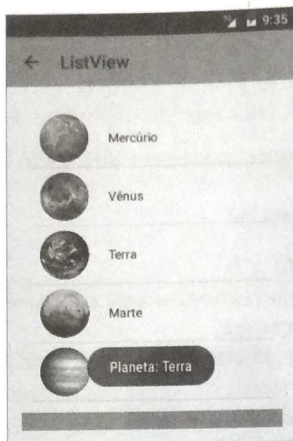


Figura 7.11 – ListView.

7.19 GridView

A classe `android.widget.GridView` é utilizada para exibir os componentes em formato de grid com linhas e colunas. Seu uso mais clássico é para exibir várias imagens como em um álbum de fotos. A seguir, podemos ver um exemplo de como utilizar um `GridView`.



/res/layout/activity_exemplo_gridview.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" android:padding="16dp" >
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Exemplo de GridView:" />
    <GridView android:id="@+id/grid1"
        android:layout_width="match_parent" android:layout_height="match_parent"
        android:padding="10dp" android:gravity="center"
        android:verticalSpacing="10dp" android:horizontalSpacing="10dp"
        android:numColumns="auto_fit" android:columnWidth="40dp" />
</LinearLayout>
```

Na tag `<GridView>` são definidos alguns parâmetros de espaçamento e número de colunas. Nesse exemplo, os parâmetros relevantes são:

Parâmetro	Descrição
columnWidth	Largura de cada coluna do grid. Sempre utilize a notação com dp (density independent pixels).
numCols	Número de colunas do grid. Nesse caso, foi informado o valor <code>auto_fit</code> para ajustar automaticamente o número de colunas com base na largura da coluna.

Para exibir as imagens no `GridView`, é necessário criar um adapter que retorne uma lista com as imagens necessárias. Sendo assim, foi definido no arquivo XML um id para o `GridView`, para que ele possa ser recuperado no código.

A seguir, temos uma classe de adapter que recebe um array de imagens e cria um `ImageView` para cada uma.



ImagemAdapter.java

```
public class ImagemAdapter extends BaseAdapter {
    private Context ctx;
    private final int[] imagens;
    public AdaptadorImagem(Context c, int[] imagens) {
        this.ctx = c;
        this.imagens = imagens;
    }
    @Override
    public int getCount() { return imagens.length; }
    @Override
    public Object getItem(int posicao) { return posicao; }
    @Override
    public long getItemId(int posicao) { return posicao; }
    @Override
    public View getView(int posicao, View convertView, ViewGroup parent) {
        // Infla a view que está no XML
        View view = LayoutInflater.from(this.ctx).inflate(R.layout.adapter_imagem_gridview,
            parent, false);
        // Utiliza o findViewById para obter o ImageView
        ImageView img = (ImageView) view.findViewById(R.id.img);
        // Altera a imagem (baseado na posição do array)
        img.setImageResource(imagens[posicao]);
        // Retorna a view
        return view;
    }
}
```

Para o código compilar, crie este arquivo XML de layout que será inflado pelo código do adapter. Veja que o nome do arquivo segue a notação `adapter_nome.xml`.

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content" android:layout_height="wrap_content" >
    <ImageView android:id="@+id/img" android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>

```

Nota: observe que o método `getView(posicao,view,parent)` deve retornar a `View` que vai ser inserida em determinada posição do `GridView`. O conceito de `adapters` (adaptadores) é muito utilizado no Android, e quanto antes você entender isso melhor. Note que para criar a `view` foi inflado um `layout XML` com ajuda da classe `LayoutInflater`.

Para finalizar o exemplo, este é o código da `activity` que vai preencher o `GridView` com um array de imagens. A classe `ImagemAdapter` deve retornar a quantidade de imagens que precisam ser adicionadas no `GridView`.

ExemploGridViewActivity.java

```

public class ExemploGridViewActivity extends Activity {
    // Array com os ids das imagens
    private int[] imagens = { R.drawable.smile1, R.drawable.smile2,
        R.drawable.smile1, R.drawable.smile2, R.drawable.smile1,
        R.drawable.smile2, R.drawable.smile1, R.drawable.smile2 };

    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        getActionBar().setDisplayHomeAsUpEnabled(true);
        setContentView(R.layout.activity_exemplo_gridview);
        GridView grid = (GridView) findViewById(R.id.grid1);
        grid.setOnItemClickListener(onGridViewItemClick());
        // Informar o adapter para preencher o GridView
        grid.setAdapter(new ImagemAdapter(this, imagens));
    }

    // Evento ao clicar no item do grid
    private OnItemClickListener onGridViewItemClick() {
        return new OnItemClickListener() {
            public void onItemClick(AdapterView<?> parent, View v, int posicao, long id) {
                Toast.makeText(ExemploGridViewActivity.this, "Imagem selecionada: " +
                    posicao, Toast.LENGTH_SHORT).show();
            }
        };
    }
}

```

Observe que o método `setOnClickListener(listener)` da classe `GridView` pode ser utilizado para tratar os eventos gerados caso o usuário selecione e pressione alguma imagem. No método `onClick(parent,view,posicao,id)`, é possível recuperar qual imagem foi selecionada. A figura 7.12 exibe o resultado desse exemplo.

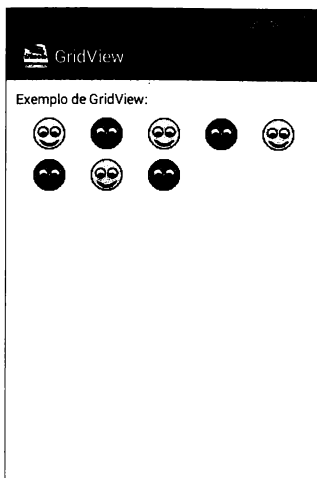


Figura 7.12 – Exemplo de GridView.

Nota: durante o livro, vamos estudar outros exemplos sobre os adapters. Se você perceber, é uma simples classe que deve implementar o método `getCount()` para informar quantas views existem, e depois o método `getView()` é chamado N vezes para criar cada view. Seja para criar um grid com `GridView` ou uma lista com `ListView`, os adapters são figurinhas carimbadas no desenvolvimento para Android e são responsáveis por fornecer o conteúdo e preencher esses componentes.

7.20 Gallery

Sabe quando você abre o álbum de fotos no Android e faz o gesto de swipe (deslizar) para os lados para ver as fotos? A classe `android.widget.Gallery` faz justamente isso. Nesse exemplo, criaremos uma galeria de imagens com as fotos de alguns planetas.

 /res/layout/activity_exemplo_gallery.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
```

```

    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Exemplo de Gallery" android:gravity="center" />
    <Gallery android:id="@+id/gallery"
        android:layout_width="match_parent" android:layout_height="match_parent"
        android:gravity="center" />
</LinearLayout>

```

A forma de usar o Gallery é idêntica à do GridView. Este próximo exemplo cria uma galeria de fotos a partir de um array de imagens.



ExemploGalleryActivity.java

```

public class ExemploGalleryActivity extends Activity {
    // Planetas
    private int[] imagens = { R.drawable.mercurio, R.drawable.venus,
        R.drawable.terra, R.drawable.marte, R.drawable.jupiter,
        R.drawable.saturno, R.drawable.urano, R.drawable.netuno,
        R.drawable.plutao };

    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_exemplo_gallery);
        Gallery g = (Gallery) findViewById(R.id.gallery);
        g.setAdapter(new ImagemAdapter(this, imagens));
        g.setOnItemClickListener(onGalleryItemClick(this));
    }

    private OnItemClickListener onGalleryItemClick(final Context context) {
        return new OnItemClickListener() {
            public void onItemClick(AdapterView<?> parent, View v, int posicao, long id) {
                // Exemplo de alerta com Toast com uma view dentro
                // Geralmente o Toast é apenas um texto
                ImageView imgView = new ImageView(context);
                imgView.setImageResource(imagens[posicao]);
                Toast t = new Toast(context);
                t.setView(imgView);
                t.show();
            }
        };
    }
}

```

Observe que, para deixar o exemplo mais completo, o método `onItemClick(parent, view, posição, id)` foi implementado para recuperar a posição da imagem selecionada. Nesse caso, a classe `Toast` foi utilizada para exibir um alerta na tela, que na verdade é desenhado pelo `ImageView` da imagem selecionada.

Nota: a mesma classe `ImageAdapter` que utilizamos para preencher as imagens do `GridView` foi utilizada agora com o `Gallery`, exatamente da mesma forma. Isso mostra a grande utilidade dos adapters. Outro detalhe importante sobre a classe `Gallery` é que ela foi recentemente descontinuada (`deprecated`) pelo Google, que recomenda utilizar a classe `ViewPager` que tem o mesmo comportamento, mas é bem mais flexível. Demonstrei o `Gallery` para seu aprendizado, mas vamos sempre usar o `ViewPager` na prática.

A figura 7.13 mostra a galeria de fotos com o planeta Terra selecionado. Para testar os exemplos, navegue na galeria fazendo o gesto de `swipe` para os lados.



Figura 7.13 – Exemplo de `Gallery`.

7.21 ViewPager

A classe `android.support.v4.view.ViewPager` faz parte da biblioteca de compatibilidade. No Android Studio, a biblioteca de compatibilidade é configurada no arquivo `app/build.gradle`.

Lembrando que a biblioteca de compatibilidade é baixada pelo **SDK Manager** pelo item **Android Support Library** e recomenda-se sempre mantê-la atualizada. Depois de


```
// Dependência da biblioteca de compatibilidade v4
compile "com.android.support:support-v4:21+"
}
```

Com a biblioteca de compatibilidade v4 configurada no projeto, vamos partir para a prática, portanto crie o seguinte arquivo de layout.

```
 /res/layout/activity_exemplo_view_pager.xml

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:gravity="center" android:text="Exemplo de ViewPager" />
    <android.support.v4.view.ViewPager android:id="@+id/viewPager"
        android:layout_width="match_parent" android:layout_height="wrap_content" />
</LinearLayout>
```

Note que a tag `<android.support.v4.view.ViewPager>` precisa conter o nome completo da classe, pois esse componente não faz parte dos nativos do Android. Para preencher as views do `ViewPager`, deve-se informar um tipo especial de `apdater`, que é uma implementação da classe `android.support.v4.view.PagerAdapter`. Lembre-se de que um `adapter` apenas retorna uma `view` para determinada posição. Dessa forma, se o `adapter` disser que existem dez páginas, ele deverá retornar uma `view` para cada uma delas.

No próximo exemplo, vamos criar a classe `ImagemPagerAdapter` que estende `android.support.v4.view.PagerAdapter` e retorna a `view` que será utilizada para criar a galeria de fotos, da mesma forma que fizemos com o `Gallery`.

 `ImagemPagerAdapter.java`

```
public class AdaptadorImagem_ViewPager extends PagerAdapter {
    private Context ctx;
    private final int[] imagens;
    public ImagemPagerAdapter(Context c, int[] imagens) {
        this.ctx = c;
        this.imagens = imagens;
    }
    @Override
    public int getCount() { // Quantidade de views do adapter
        return imagens != null ? imagens.length : 0;
    }
}
```

```

@Override
public Object instantiateItem(ViewGroup container, int position) {
    // Infla a view
    View view = LayoutInflater.from(this.ctx).inflate(R.layout.adapter_imagem,
        container, false);
    ImageView img = (ImageView) view.findViewById(R.id.img);
    img.setImageResource(imagens[position]);
    ((ViewGroup) container).addView(view); // Adiciona ao layout ViewGroup
    return view;
}

@Override
public void destroyItem(ViewGroup container, int position, Object view) { // Remove a
                                                                    // view do container
    ((ViewPager) container).removeView((View) view);
}

@Override
public boolean isViewFromObject(View view, Object object) {
    // Determina se a view informada é igual ao object retornado pelo instantiateItem
    return view == object;
}
}

```

Para auxiliar o entendimento do código-fonte dessa classe, segue a explicação de cada método:

Método	Descrição
<code>getCount</code>	Retorna a quantidade de elementos do adapter.
<code>instantiateItem</code>	Retorna um objeto-chave chamado de key object, que é utilizado internamente para controlar o ViewPager. Durante a implementação do método <code>instantiateItem</code> , você deve adicionar a view criada no container, que é o ViewGroup informado como parâmetro. Para simplificar, você pode retornar diretamente a view neste método ou outro objeto-chave qualquer.
<code>isViewFromObject</code>	Nesse método, você deve validar se a view informada como parâmetro corresponde ao objeto-chave informado. Esse objeto-chave é aquele que foi retornado do método <code>instantiateItem</code> .
<code>destroyItem</code>	Este método é chamado para destruir uma view associada a um objeto-chave. Portanto, as views que foram adicionadas ao container devem ser removidas aqui. É importante que esse método seja sobrescrito e o <code>super.destroyItem()</code> da classe-mãe não seja chamado. Ao navegar pelo ViewPager, o Android pode ir destruindo as views que não estão sendo utilizadas para economizar memória.

Vamos finalizar o exemplo e escrever o código da activity que vai utilizar o ViewPager.

**ExemploViewPagerActivity.java**

```

public class ExemploViewPagerActivity extends Activity {
    // Planetas
    private int[] imagens = { R.drawable.mercurio, R.drawable.venus,
        R.drawable.terra, R.drawable.marte, R.drawable.jupiter,
        R.drawable.saturno, R.drawable.urano, R.drawable.netuno, R.drawable.plutao };

    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_exemplo_view_pager);
        ViewPager g = (ViewPager) findViewById(R.id.viewPager);
        g.setAdapter(new ImagePagerAdapter(this, imagens));
        g.setOnPageChangeListener(new OnPageChangeListener() {
            @Override
            public void onPageSelected(int position) { // Informa que determinada página
                // foi selecionada
                Toast t = Toast.makeText(getBaseContext(), "Imagem: " + position,
                    Toast.LENGTH_SHORT);
                t.show();
            }
            @Override
            public void onPageScrolled(int position, float positionOffset,
                int positionOffsetPixels) {
            }
            @Override
            public void onPageScrollStateChanged(int state) {
            }
        });
    }
}

```

Ao executar esse exemplo, o funcionamento será idêntico ao do componente Gallery. Note que, ao navegar de uma página para outra no ViewPager, um alerta é exibido com o Toast, porque o método `onPageSelected(position)` é chamado. Dessa forma, podemos realizar alguma ação sempre que o usuário navegar entre as views desse componente.

A classe ViewPager é uma das mais utilizadas no desenvolvimento para Android, e se integra muito bem com a famosa API de Fragments que ainda vamos estudar durante o livro. Lembre-se de que a classe Gallery foi descontinuada pelo Google, e o recomendado é sempre utilizar o ViewPager.

Provavelmente você já deve ter visto alguns aplicativos que usam tabs, como por exemplo: o Google Play, o qual permite navegar pelas tabs utilizando os gestos de swiipe lateral. Isso também é feito com o `ViewPager`, inclusive vamos fazer exatamente isso ao desenvolver o aplicativo dos carros.

Dica: uma biblioteca muito conhecida por desenvolvedores Android é a `ViewPagerIndicator` (<http://viewpagerindicator.com>). Ela funciona em conjunto com o `ViewPager` e contém várias views utilitárias. Por exemplo, é bem comum em aplicativos ter aquele marcador com bolinhas em baixo do `ViewPager`, para indicar a página que você está visualizando. Essa biblioteca contém esse marcador e muito mais.

7.22 ViewPager + TitleStrip ou TabStrip

No exemplo anterior, mostramos a figura de cada planeta no `ViewPager`, mas não mostramos o nome do planeta em lugar nenhum.

Para mostrar o nome do planeta, podemos implementar a interface `OnPageChangeListener` para monitorar a troca de páginas do `ViewPager`, e nesse caso mostrar o nome do planeta em algum `TextView`. Mas o `ViewPager` já tem duas classes que facilitam justamente esse trabalho, a classe `android.support.v4.view.PagerTitleStrip` e `android.support.v4.view.PagerTabStrip`.

Para ter uma ideia do que estou falando, a figura 7.15 mostra o resultado ao utilizar o `ViewPager` com o `PagerTitleStrip`.



Figura 7.15 – `PagerTitleStrip` e `PagerTabStrip`.

A classe `android.support.v4.view.PagerTitleStrip` mostra um título acima do `ViewPager` facilitando o entendimento do usuário. Já a classe `android.support.v4.view.PagerTabStrip` mostra um indicador com uma tab e inclusive permite ao usuário clicar na tab para navegar entre as páginas. Para utilizar um desses componentes, basta incluí-lo como filho do `ViewPager` no layout. O exemplo a seguir mostra como utilizar o `PagerTabStrip`.

 `/res/layout/activity_exemplo_view_pager_tab_strip.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout . . . >
    <android.support.v4.view.ViewPager android:id="@+id/viewPager"
        android:layout_width="match_parent" android:layout_height="wrap_content" >
        <android.support.v4.view.PagerTabStrip android:id="@+id/viewPagerTabStrip"
            android:layout_width="match_parent" android:layout_height="wrap_content"
            android:layout_gravity="top" android:background="#33b5e5"
            android:textColor="#fff" />
        </android.support.v4.view.ViewPager>
</LinearLayout>
```

Para mostrar o título acima do `ViewPager`, é preciso que a classe do adapter implemente o método `getPageTitle(int page)` que deve retornar o título da página. Nesse exemplo, vamos usar novamente a classe `Planeta`, a qual tem o nome do planeta e o inteiro com o recurso da imagem. A classe do adapter recebe uma lista de planetas e retorna a view que contém a foto do planeta, também retorna o nome do planeta no método `getPageTitle(int page)`.

 `PlanetasPagerAdapter.java`

```
public class PlanetasPagerAdapter extends PagerAdapter {
    private Context ctx;
    private final List<Planeta> planetas;
    public PlanetasPagerAdapter(Context c, List<Planeta> planetas) {

    }
    // Para o código completo veja nos exemplos do livro. Esse é um adapter simples.
    // O importante é o método getPageTitle(page) retornar o título.
    @Override
    public CharSequence getPageTitle(int page) {
        // Título para mostrar no PagerTitleStrip ou PagerTabStrip
        Planeta planeta = planetas.get(page);
        return planeta.nome;
    }
}
```

O código da activity somente cria a lista de planetas e configura o adapter no ViewPager. O resto é tudo automático, e o título retornado pelo adapter será mostrado na tab do ViewPager.

 MainActivity.java

```
public class ExemploViewPagerTabStripActivity extends Activity {
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_exemplo_view_pager_tab_strip);
        getActionBar().setDisplayHomeAsUpEnabled(true);
        // Planetas
        List<Planeta> planetas = Planeta.getPlanetas();
        // ViewPager
        ViewPager g = (ViewPager) findViewById(R.id.viewPager);
        g.setAdapter(new PlanetasPagerAdapter(this, planetas));
    }
}
```

Dica: para alterar por programação a página que o ViewPager está mostrando, utilize o método `setCurrentItem(int page)`.

7.23 ImageSwitcher

Outra classe bastante útil é a `android.widget.ImageSwitcher`, utilizada para mostrar uma imagem após outra de forma animada.

 /res/layout/activity_exemplo_image_switcher.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" android:padding="10dp" >
    <Button android:id="@+id/btProxima"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/proxima" />
    <ImageSwitcher android:id="@+id/imageSwitcher"
        android:layout_width="match_parent" android:layout_height="match_parent"
        android:layout_margin="10dp" />
</LinearLayout>
```

**ExemploImageSwitcherActivity.java**

```

public class ExemploImageSwitcherActivity extends Activity {
    // Planetas
    private int[] imagens = { R.drawable.mercurio, R.drawable.venus,
        R.drawable.terra, R.drawable.marte, R.drawable.jupiter,
        R.drawable.saturno, R.drawable.urano, R.drawable.netuno, R.drawable.plutao };
    private ImageSwitcher imageSwitcher;
    private int idx = 0;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getActionBar().setDisplayHomeAsUpEnabled(true);
        setContentView(R.layout.activity_exemplo_image_switcher);
        // Configura o ImageSwitcher e os efeitos
        imageSwitcher = (ImageSwitcher) findViewById(R.id.imageSwitcher);
        imageSwitcher.setFactory(new ImageSwitcher.ViewFactory(){
            @Override
            public View makeView() {
                ImageView img = new ImageView(getBaseContext());
                img.setScaleType(ImageView.ScaleType.FIT_CENTER);
                img.setLayoutParams(new ImageSwitcher.LayoutParams(LayoutParams.MATCH_PARENT,
                    LayoutParams.MATCH_PARENT));
                return img;
            }
        });
        imageSwitcher.setInAnimation(AnimationUtils.loadAnimation(this, android.R.anim.fade_in));
        imageSwitcher.setOutAnimation(AnimationUtils.loadAnimation(this, android.R.anim.fade_out));
        View btProxima = findViewById(R.id.btProxima);
        btProxima.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View arg0) {
                if(idx == imagens.length) { idx = 0; }
                imageSwitcher.setImageResource(imagens[idx++]);
            }
        });
    }
}

```

Para a classe ImageSwitcher funcionar, é necessário chamar o método `setFactory(viewFactory)` informando uma implementação de `android.widget.ViewSwitcher.ViewFactory`. Essa interface define o método `makeView()` que deve retornar uma View, que é a imagem que deve ser exibida. Ao chamar o método `setImageResource(imagem)` no ImageSwitcher,

a animação é realizada. Neste exemplo a imagem é alterada sempre que o botão **Próxima** for clicado, conforme a figura 7.16.



Figura 7.16 – Exemplo de ImageSwitcher.

7.24 WebView

Se por algum motivo você precisar exibir uma página web dentro do aplicativo, a classe `android.webkit.WebView` pode ser útil. O funcionamento desse componente é idêntico ao browser do Android, isso porque internamente é utilizada a engine WebKit.

Essa é uma das views mais utilizadas nos aplicativos, principalmente pelos adeptos da criação de aplicativos híbridos que utilizam HTML5 e JavaScript para criar a interface gráfica. Primeiramente, para ter acesso à internet, declare a permissão `INTERNET` no arquivo `AndroidManifest.xml`. Quando o usuário baixar o aplicativo da Google Play, serão mostradas para ele todas as permissões que o aplicativo precisa utilizar, e baseado nisso o usuário pode aprovar ou não a instalação.



AndroidManifest.xml

```
<manifest ... />
    <uses-permission android:name="android.permission.INTERNET" />
    <application ... />
</manifest>
```

Para demonstrar como exibir uma página de internet usando o `WebView`, vamos criar um simples exemplo que exibe a página www.livroandroid.com.br.

 /res/layout/activity_exemplo_webview.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=http://schemas.android.com/apk/res/android" ... />
    <FrameLayout android:layout_width="match_parent" android:layout_height="match_parent" >
        <WebView android:id="@+id/webview" android:layout_margin="10dp"
            android:layout_width="match_parent" android:layout_height="match_parent" />
        <ProgressBar android:id="@+id/progress" android:layout_gravity="center"
            android:layout_width="wrap_content" android:layout_height="wrap_content" />
    </FrameLayout>
</LinearLayout>
```

 ExemploWebViewActivity.java

```
public class ExemploWebViewActivity extends Activity {
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_exemplo_webview);
        final WebView webView = (WebView) findViewById(R.id.webview);
        final View progress = findViewById(R.id.progress);
        progress.setVisibility(View.INVISIBLE);
        webView.loadUrl("http://www.livroandroid.com.br");
        webView.setWebViewClient(new WebViewClient(){
            @Override
            public void onPageStarted(WebView view, String url, Bitmap favicon) {
                progress.setVisibility(View.VISIBLE); // página começou a ser carregada
            }
            @Override
            public void onPageFinished(WebView view, String url) {
                progress.setVisibility(View.INVISIBLE); // página foi carregada
            }
            @Override
            public void onReceivedError(WebView view, int errorCode, String description,
                String failingUrl) {
                // Erro ao carregar a página do WebView (endereço errado, ou erro de conexão)
            }
        });
    }
}
```

A figura 7.17 mostra a página do site do livro aberta dentro do WebView. Veja que utilizamos o `FrameLayout` para inserir o `ProgressBar` por cima do `WebView`. Enquanto a página está sendo carregada, o `ProgressBar` fica visível para mostrar a animação.

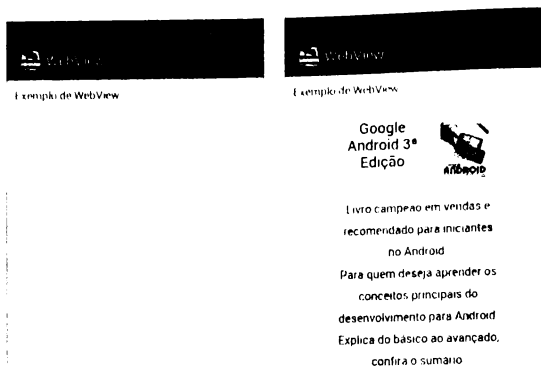


Figura 7.17 – Exemplo do WebView.

Nota: lembre-se de que é necessário declarar a permissão `INTERNET` para o `WebView` funcionar.

O `WebView` realmente é muito simples de ser utilizado. Nem vou me estender muito nos exemplos porque é muito fácil encontrar material sobre o `WebView`. Apenas para ter uma ideia, você até pode enviar um `HTML` em `string` e pedir para o `WebView` renderizar a página.

```
webView.loadDataWithBaseURL("", "<font color='blue'>HTML aqui<font>",  
    "text/html", "UTF-8", "");
```

Geralmente os aplicativos deixam esse código `HTML` em arquivos texto dentro do projeto, então basta ler o arquivo e converter para `string`. Outra técnica que é muito utilizada é injetar código `JavaScript` no `WebView`, para os mais variados fins, e isso pode ser feito assim:

```
webView.getSettings().setJavaScriptEnabled(true);  
webView.loadUrl("javascript:alert('Oi leitor');");
```

Essas técnicas de injetar código `HTML` e `JavaScript` no `WebView` estão fazendo a festa dos adeptos de `HTML5` e aplicativos híbridos, e é com base nelas que populares frameworks como o `PhoneGap` funcionam. Esse tipo de abordagem permite que desenvolvedores com vasta experiência em web entrem no mercado dos aplicativos móveis, e utilizem código `HTML` para criar a interface gráfica.

Mas não estou aqui para discutir sobre este assunto nativo vs. `HTML`. O objetivo deste livro é ensinar a criar aplicativos nativos para `Android`, além de explicar os principais conceitos da plataforma.

7.25 Movimentando uma imagem pela tela com touch

Neste próximo exemplo, vamos movimentar uma imagem pela tela, utilizando touch screen. No emulador você poderá utilizar o mouse para movimentar a imagem e naturalmente em um celular real você vai arrastar a imagem com o dedo.

A classe View contém o método `onTouchEvent(MotionEvent)`, que sempre é chamado quando um toque na tela é realizado. Como parâmetro temos um objeto do tipo `MotionEvent`, com o qual é possível recuperar as posições `x` e `y` do toque.

```
public boolean onTouchEvent(MotionEvent event) {  
    float x = event.getX();  
    float y = event.getY();  
    return true;  
}
```

Esse método deve retornar `true` caso a view tenha tratado o evento, ou `false` se é para delegar a tarefa para as outras views da tela. Se nenhuma view tratar o evento, o mesmo método será chamado na activity responsável pela tela. A seguir podemos ver um exemplo completo que permite mover a imagem do boneco do Android pela tela. A figura foi inserida em `/res/drawable/android.png`.

TouchScreenView.java

```
public class TouchScreenView extends View {  
    private static final String TAG = "livro";  
    private Drawable img;  
    int x, y;  
    private boolean selecionou;  
    private int larguraTela;  
    private int alturaTela;  
    private int larguraImg;  
    private int alturaImg;  
    public TouchScreenView(Context context) {  
        super(context, null);  
        // Recupera a Imagem  
        img = context.getResources().getDrawable(R.drawable.android);  
        // Recupera a largura e altura da imagem  
        larguraImg = img.getIntrinsicWidth();  
        alturaImg = img.getIntrinsicHeight();  
        // Configura a View para receber foco e tratar eventos de teclado  
        setFocusable(true);  
    }  
    @Override
```

```

// Chamado quando a tela é redimensionada, ou iniciada...
protected void onSizeChanged(int width, int height, int oldw, int oldh) {
    super.onSizeChanged(width, height, oldw, oldh);
    this.larguraTela = width;
    this.alturaTela = height;
    x = width / 2 - (larguraImg / 2);
    y = height / 2 - (alturaImg / 2);
    Log.i(TAG, "onSizeChanged x/y: " + x + "/" + y);
}

@Override
// Desenha a tela
public void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    // Fundo branco
    Paint pincel = new Paint();
    pincel.setColor(Color.WHITE);
    canvas.drawRect(0, 0, larguraTela, alturaTela, pincel);
    // Define os limites/área para desenhar
    img.setBounds(x, y, x + larguraImg, y + alturaImg);
    // Desenha a imagem
    img.draw(canvas);
}

@Override
// Move a imagem
public boolean onTouchEvent(MotionEvent event) {
    float x = event.getX();
    float y = event.getY();
    Log.i(TAG, "onTouchEvent: x/y > " + x + "/" + y);
    switch (event.getAction()) {
        case MotionEvent.ACTION_DOWN:
            // Inicia o movimento se pressionou a imagem
            selecionou = img.getBounds().contains((int) x, (int) y);
            break;
        case MotionEvent.ACTION_MOVE:
            // Arrasta o boneco
            if (selecionou) {
                this.x = (int) x - (larguraImg / 2);
                this.y = (int) y - (alturaImg / 2);
            }
            break;
        case MotionEvent.ACTION_UP:
            // Finaliza o movimento
            selecionou = false;
    }
}

```

```

        break;
    }
    // O invalidate vai chamar o método onDraw(canvas) novamente
    invalidate();
    return true;
}
}

```

Feito isso, crie uma activity e configure esta view no método `setContentView(view)`:



TouchScreenViewActivity.java

```

public class TouchScreenViewActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(new TouchScreenView(this));
    }
}

```

Nota: neste exemplo, ao tocar na tela a coordenada x/y da figura é atualizada. Ao chamar o método `invalidate()`, o Android vai desenhar a view novamente chamando o método `onDraw(canvas)`.

Ao testar o exemplo no emulador, utilize o mouse para simular o touch screen e mover a imagem (Figura 7.18).

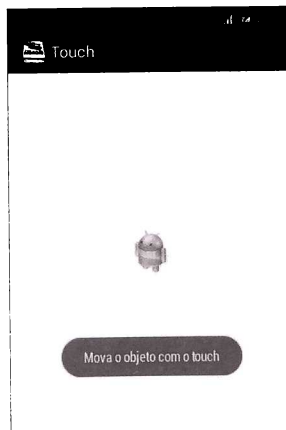


Figura 7.18 – Exemplo de touch screen.

7.26 Desenho manual com a classe Canvas

O Android tem várias classes prontas para desenhar componentes na tela e cada uma dessas classes é uma subclasse de `View`. Será que podemos criar nossa própria classe-filha de `View`? A resposta é sim.

Neste próximo exemplo, criaremos um componente customizado que será uma subclasse direta de `android.view.View` e aprenderemos a desenhar quadrados, círculos e linhas coloridas na tela. Para isso, sobrescreveremos o método `onDraw(Canvas)`, o qual é chamado pelo Android para desenhar um componente (`view`). As classes `TextView`, `EditText`, `ImageView`, `Button` etc. já implementam esse método, mas chegou a hora de criarmos nosso próprio componente do zero e desenhar tudo na tela.

Para isso criaremos uma classe chamada `MinhaView`, conforme demonstrado a seguir:

`MinhaView.java`

```
package br.com.livroandroid.cap07_view.canvas;

import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.util.AttributeSet;
import android.view.View;

public class MinhaView extends View {
    // Para definir a cor RGB
    private Paint pincelVermelho;
    private Paint pincelPreto;
    private Paint pincelAzul;
    public MinhaView(Context context) {
        this(context,null);
    }
    public MinhaView(Context context, AttributeSet attrs) {
        super(context, attrs);
        setBackgroundColor(Color.LTGRAY);
        // Vermelho
        pincelVermelho = new Paint();
        pincelVermelho.setARGB(255, 255, 0, 0);
        // Preto
        pincelPreto = new Paint();
        pincelPreto.setARGB(255, 0, 0, 0);
        // Azul
        pincelAzul = new Paint();
        pincelAzul.setARGB(255, 0, 0, 255);
    }
}
```

```

        // Configura a View para receber foco e tratar eventos de teclado
        setFocusable(true);
    }

    @Override
    public void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        // Desenha um quadrado
        canvas.drawRect(20, 20, 200, 200, pincelAzul);
        // Desenha uma linha
        canvas.drawLine(200, 200, 400, 400, pincelPreto);
        // Desenha um círculo
        canvas.drawCircle(400, 400, 100, pincelVermelho);
    }
}

```

Observe que existem dois construtores na classe: um que recebe apenas o `android.content.Context` e outro que também recebe o `android.util.AttributeSet`. Se essa classe for utilizada diretamente pela API Java, o construtor com apenas o parâmetro `android.content.Context` é chamado; caso contrário, se for utilizada pelo XML, o Android chamará o construtor com os dois parâmetros. Se for necessário, a classe `android.util.AttributeSet` é utilizada para ler os parâmetros definidos no XML.

A classe `MinhaView` implementa o método `onDraw(Canvas)` e desenha um retângulo, uma linha e um círculo, utilizando os métodos da classe `android.graphics.Canvas`. Além das coordenadas `x` e `y` para desenhar as formas geométricas, foi criado um objeto do tipo `android.graphics.Paint`, o qual define as cores do desenho. Para isso, foram criados três objetos `Paint` (pincel) e para cada um foi atribuída uma cor RGB diferente.

Para utilizar essa nova classe em algum arquivo de layout, basta inserir uma tag com seu nome completo, neste caso: `<br.com.livroandroid.cap07_view.MinhaView>`.

 `/res/layout/activity_exemplo_minha_view.xml`

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent" >
    <br.com.livroandroid.cap07_view.canvas.MinhaView
        android:id="@+id/canvas"
        android:layout_width="match_parent" android:layout_height="match_parent" />
</LinearLayout>

```


A figura 7.19 mostra a pré-visualização dessa tela pelo editor, mesmo com um componente desenhado manualmente. Como a pré-visualização da tela já é o suficiente para entender o exemplo, não criaremos aqui nenhuma tela/activity que utilize esse arquivo de layout. Isso fica como exercício para você.



Figura 7.19 – Componente customizado desenhado manualmente.

7.27 Nunca utilize pixels

Já foi falado anteriormente para nunca utilizar pixels ao definir tamanhos de views e espaçamentos em arquivos de layout XML. O mesmo conceito vale ao desenhar códigos utilizando a API de Canvas.

No exemplo anterior, utilizamos o seguinte código para desenhar um quadrado:

```
// Desenha um quadrado  
canvas.drawRect(20, 20, 200, 200, pincelAzul);
```

No entanto, esse código que desenha o quadrado tem um problema, pois está usando valores fixos em pixels. Isso vai trazer resultados diferentes em telas que apresentem diferentes resoluções e densidade. O correto seria utilizar valores em dp (density independent pixel) e converter o valor para pixel utilizando a densidade de cada tela. Por exemplo, 100px pode ser convertido para 75px, 100px, 150px, 200px, 300px, conforme a densidade da tela do aparelho. Essas conversões precisam ser feitas no código para garantir que o resultado será o mesmo independentemente da resolução e densidade da tela.

A seguir, podemos ver uma função que converte o valor de dp para pixel.

```
// Converte um valor em dp para pixels
public float toPixels(float dip) {
    Resources r = getContext().getResources();
    float densidade = r.getDisplayMetrics().density; // Densidade da tela
    int px = (int) (dip * densidade + 0.5f);
    return dip;
}
```

Portanto, o correto para desenhar um quadrado é utilizar um código assim:

```
// Desenha um quadrado
canvas.drawRect(toPixels(20), toPixels(20), toPixels(200), toPixels(200), pincelAzul);
```

Acredito que talvez seja cedo para explicar por que isso é necessário. Prefiro continuar o livro com conceitos simples e focar no que você vai utilizar no dia a dia. Ainda temos alguns capítulos essenciais pela frente e logo vamos começar o desenvolvimento do aplicativo dos carros.

A metodologia do livro é explicar cada conceito de uma vez. Quero focar nos conceitos principais, aqueles de que, tenho certeza, você vai precisar no dia a dia.

Conheço muitos desenvolvedores que fazem aplicativos e não sabem explicar o que é dp (density independent pixel), por isso creio que esse assunto possa ficar para depois. Para mais detalhes, leia o capítulo 30, sobre como suportar diferentes tamanhos de telas.



CAPÍTULO 8

Fragments

Um fragment é um componente de código reutilizável, responsável por criar sua própria view, tratar os eventos e gerenciar o seu próprio conteúdo.

Uma activity pode conter um ou mais fragments que podem ser adicionados no layout como se fossem views, porém os fragments têm comportamento próprio e são autogerenciáveis.

8.1 Como surgiram os fragments no Android 3.0 Honeycomb

Com a popularização dos tablets e a grande busca dos usuários por esses dispositivos, surgiu a necessidade de otimizar e customizar o Android para usufruir ao máximo do tamanho de tela disponível nesses dispositivos. Foi assim que surgiu o Android 3.0 Honeycomb, a primeira versão do Android otimizada para tablets.

E com o Honeycomb, nasceu a API de fragments, originalmente utilizada para organizar a grande tela dos tablets em pequenos componentes, mas aos poucos todos perceberam que fragments eram muito mais do que isso.

Ao desenvolver para smartphones, geralmente temos uma tela simples, pois o espaço disponível é limitado. Dessa forma, o modelo tradicional com uma activity e uma view, no qual a activity controla toda a lógica da tela, sempre atendeu as necessidades. Mas criar aplicações para tablets é uma arte, e vários fatores precisam ser levados em consideração. O mais importante de todos é o tamanho da tela, que deve ser aproveitado ao máximo. Muitas vezes, é necessário preencher a tela com várias views, cada uma com um conteúdo diferente.

A figura 8.1 compara o modelo tradicional de uma aplicação do tipo lista e detalhes, executando no smartphone ou tablet. No smartphone, duas telas precisam ser utilizadas para fazer a navegação da lista para a tela de detalhes. No tablet, podemos utilizar uma única tela, aproveitando ao máximo o espaço disponível.

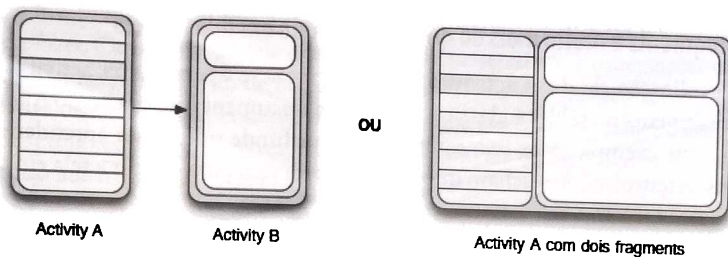


Figura 8.1 – Fragment que divide a tela em pedaços.

Dica: se quiser ver um aplicativo com duas telas de lista e detalhes, abra o projeto Planetas-Activity no Android Studio. Neste projeto, temos uma lista de planetas e ao clicar na lista o planeta é mostrado em outra tela. Durante este capítulo, vamos trabalhar em cima deste exemplo.

Nesta figura podemos ver que a aplicação para smartphone utiliza a Activity A e a Activity B como de costume. Na aplicação para tablet, como existe somente uma tela, ou seja, apenas uma activity, dentro dela o conteúdo é separado em dois fragments. Basicamente um fragment é um componente que pode ser inserido dentro da activity, e esse componente fica responsável por:

1. Criar a view para preencher determinado espaço, além de controlar o seu estado e tratar os eventos.
2. Pela lógica de negócios para buscar os dados de um web service ou banco de dados.
3. Por atualizar a sua view de forma independente da activity e de qualquer outro fragment da tela.

Eu costumo dizer que um fragment é uma *miniactivity*, que tem sua própria view e lógica, além de ser responsável por gerenciar seu próprio conteúdo.

Nota: talvez este capítulo seja um pouco avançado neste momento, tudo depende do seu perfil. O fato é que antes de começarmos a desenvolver o aplicativo dos carros eu preciso explicar o que é um Fragment. E como vou ter de explicar o assunto, aproveito para abordar alguns tópicos mais avançados, como tablets e gerenciamento de estado do fragment. Mesmo que o assunto seja um pouco avançado, continue estudando, pois o livro servirá de consulta mais tarde.

8.2 Fragments é muito mais do que dividir a tela em duas partes

Esta explicação de duas activities no smartphone e apenas uma activity com dois fragments no tablet é clássica, inclusive a documentação oficial do Android utiliza este exemplo. Mas isso muitas vezes confunde quem está aprendendo, e alguns desenvolvedores acham que fragments servem para dividir a tela em duas partes nos tablets.

Na verdade, os fragments são componentes que ficam espalhados na tela, sendo que um de seus principais objetivos é reutilizar a lógica entre a versão **smartphone** e tablet. Para começarmos bem, vamos acabar com o mito de que um **fragment** serve para dividir a tela do tablet em duas partes. A figura 8.2 mostra algo diferente, a tela do tablet dividida em três partes.

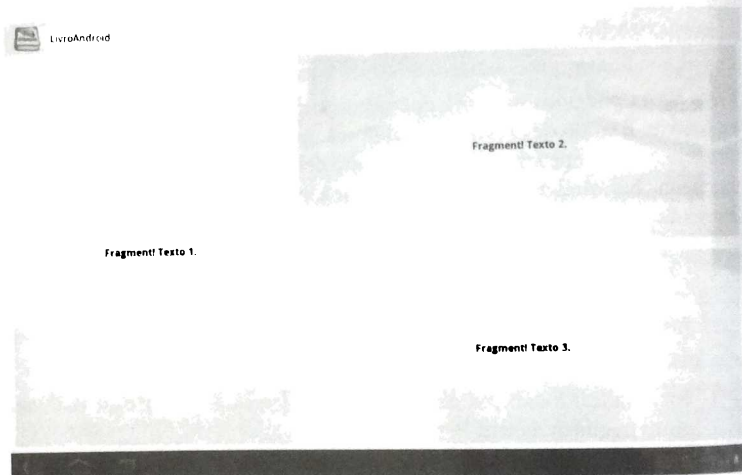


Figura 8.2 – Activity dividida em três partes com fragments.

Esse tipo de tela é bem comum nos tablets, pois assim você consegue mostrar várias informações ao mesmo tempo para o usuário. A principal razão de quebrar a tela em pedaços é para simplificar o código da activity, pois podemos dizer que cada fragment é responsável por determinada parte do layout.

A figura 8.3 mostra um dos meus primeiros projetos para tablets, quando estudei fragments pela primeira vez. Neste aplicativo, cada pedaço da tela é um fragment, como a lista de índices, lista de notícias, lista de vídeos, área central, gráfico etc. Os benefícios dessa organização são imensos, pois os fragments deixam o

código muito mais limpo e organizado, pois cada componente faz apenas o que tem de fazer. Cada parte da tela é um componente separado e independente dos outros. Por exemplo, a lista de índices busca os dados em um web service e cria o ListView de forma independente do resto da tela. No final, o código da activity fica bastante reduzido, ou talvez vazio, pois ela não faz nada, e todo o trabalho é delegado aos fragments.

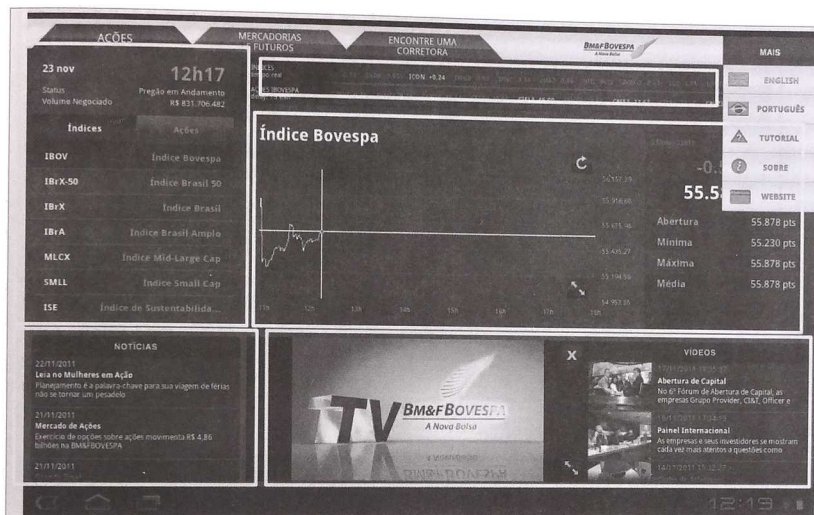


Figura 8.3 – Dividindo a tela em fragments.

A figura 8.4 mostra outro aplicativo para tablet, em que mais uma vez este conceito de dividir a tela em partes foi utilizado. Cada parte é um fragment, e novamente isso ajudou a separar as responsabilidades e organizar o código. Se um aplicativo desse tamanho fosse criado apenas com uma activity, o XML de layout seria imenso e o código-fonte para gerenciar toda a lógica também. No entanto, com fragments a activity apenas divide o seu layout em partes e insere cada fragment no seu devido espaço. Cada fragment por sua vez vai criar a view e gerenciar o conteúdo. Simples, não é?

Trabalhar com fragments, porém, é muito mais do que apenas separar a tela em pedaços, pois um dos principais objetivos da API é criar um componente de código reutilizável. A figura 8.5 mostra uma aplicação compatível com smartphones e tablets. A parte esquerda da figura mostra a versão para tablets. Veja que o tablet está mostrando a lista de índices em determinado local da tela. E na parte da direita podemos ver a mesma aplicação no smartphone, com a mesma lista

de índices. Este é um exemplo no qual os fragments são utilizados para criar componentes reutilizáveis, facilitando a manutenção de ambas as versões. Toda a lógica para buscar os dados do web service fica encapsulada no fragment, e no máximo o fragment precisa customizar o layout para se adequar às telas do smartphone ou tablet.



Figura 8-4 Dividindo a tela em fragments

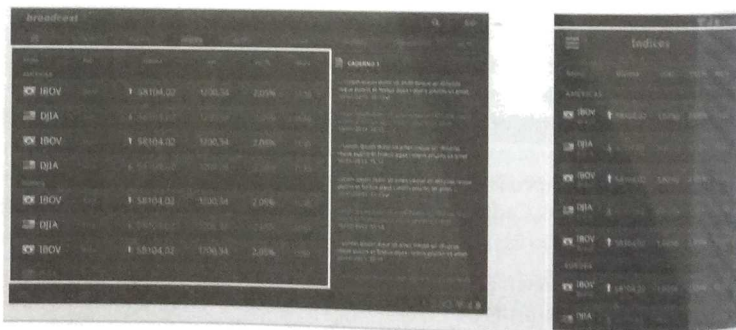


Figura 8-5 Reutilizando componentes entre a versão tablet e smartphone.

Atualmente os fragments também são muito utilizados no desenvolvimento de aplicativos modernos para smartphones, e a figura 8.6 mostra o aplicativo de

Google Play que utiliza tabs na action bar. Ao selecionar uma tab, o conteúdo do centro é atualizado, mas se você reparar não é feita a navegação entre telas. Já aprendemos no livro que, sempre que for aberta uma nova tela, é necessário criar uma nova activity, mas neste caso o aplicativo permanece na mesma tela, pois é necessário trocar apenas o conteúdo central que mostra a tab selecionada.

Por isso, ao utilizar tabs na action bar, o conteúdo da tela precisa ser um fragment, pois um fragment é um componente que pode ser inserido, removido e substituído de determinado local do layout. Ao selecionar uma tab, basicamente um fragment é substituído por outro.



Figura 8.6 – Aplicativo do Google Play.

A figura 8.7 mostra o padrão de navegação **Navigation Drawer**, muito utilizado nos aplicativos modernos. Só para dar uma ideia, o Gmail e YouTube utilizam esse padrão, que é conhecido popularmente por “menu lateral”. Tal padrão de navegação, da mesma forma que as tabs, representam a navegação top-level do aplicativo com as seções mais comuns. Nesse caso existe apenas uma activity, pois, ao abrir o menu lateral e selecionar uma opção, apenas o conteúdo central é alterado, sem fazer a navegação de telas. Na prática, ao selecionar uma opção do menu, um fragment é substituído por outro.

A figura 8.8 mostra outro caso no qual podemos utilizar um fragment, para solucionar o problema de inserir o banner na parte inferior da tela. Como esse banner é um componente e deve ser inserido em todas as telas, o ideal é que ele seja um fragment, pois basta inseri-lo no layout. O fragment sozinho vai criar a view e buscar o conteúdo necessário para desenhar o banner.

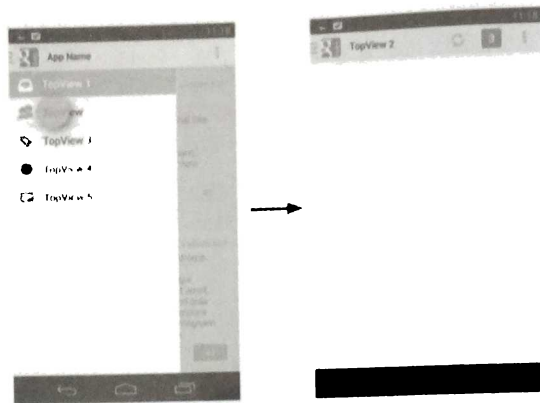


Figura 8.7 -- Menu com navigation drawer.



Figura 8.8 – Aplicativo com anúncios.

Nota: estes são alguns exemplos nos quais podemos utilizar a API de fragments, seja para organizar o código do tablet, smartphone ou para reutilizar componentes. Atualmente, mesmo se você criar uma nova activity, é recomendado encapsular o código dessa activity em um fragment; assim, se um dia você precisar reaproveitar esse componente, o código já estará pronto.

8.3 API de Fragments

Antes de começarmos a estudar os exemplos com código-fonte, vamos aprender as principais classes da API de Fragments.

`android.app.Fragment`

Classe que o fragment deve estender. É necessário sobrescrever o método `onCreateView(inflater, container, bundle)` para criar a view.

`android.app.FragmentManager`

Classe que gerencia os fragments pela API. Contém os métodos `findFragmentById(id)` e `findFragmentByTag(tag)` utilizados para encontrar os fragments no layout, de forma similar ao método `findViewById(id)` que uma `activity` utiliza para buscar uma view.

`android.app.FragmentTransaction`

Classe utilizada para adicionar, remover ou substituir os fragments dinamicamente no layout.

Essas três classes só podem ser utilizadas no Android 3.0 ou superiores, por isso foi criada a biblioteca de compatibilidade v4, compatível com Android 1.6 (API Level 4) ou superior. Para configurar a dependência para a biblioteca v4, basta adicionar uma linha no arquivo *app/build.gradle*.



`app/build.gradle`

```
...
dependencies {
    ...
    // Dependência da biblioteca de compatibilidade v4
    compile "com.android.support:support-v4:21+"
}
```

Feito isso, podemos utilizar as classes da biblioteca de suporte que ficam no pacote `android.support.v4`. Para manter a compatibilidade com versões anteriores, recomenda-se utilizar as seguintes classes, no lugar das nativas.

- `android.support.v4.app.Fragment`
- `android.support.v4.app.FragmentManager`
- `android.support.v4.app.FragmentTransaction`

A biblioteca de compatibilidade é distribuída pelo SDK Manager e pode receber atualizações, por isso recomendo que você sempre verifique se existem versões novas da biblioteca.

Nota: neste livro, todos os exemplos de fragments utilizam a classe `android.support.v4.app.Fragment`; lembre-se desse detalhe ao escrever o código e testar os exemplos.

Para fechar o assunto, falta um pequeno detalhe. A classe `FragmentManager` é uma das principais classes da API e ela é recuperada com o seguinte código dentro de uma activity ou fragment.

```
android.app.FragmentManager fm = getFragmentManager();
```

O método `getFragmentManager()` retorna a versão nativa da classe, então não podemos utilizá-lo. Por isso, todas as activities do projeto devem estender `android.support.v4.app.FragmentActivity` ou `android.support.v7.app.AppCompatActivity`, que contém o método `getSupportFragmentManager()`, o qual retorna a classe de compatibilidade.

```
android.support.v4.app.FragmentManager fm = getSupportFragmentManager();
```

Nota: a classe `FragmentActivity` é mãe de `AppCompatActivity`. Portanto, sempre que você utilizar a classe `AppCompatActivity` para utilizar a action bar de compatibilidade, ganhamos de brinde o acesso à biblioteca dos fragments.

8.4 Hello World fragment

Para ficar mais fácil de entender o assunto, vamos ver um pouco de código-fonte e brincar um pouco com os fragments. Crie o projeto com o nome **HelloFragments**, com a activity `MainActivity` e o template **Blank Activity**.

Na activity principal não vamos fazer nada, e seu código será simples conforme demonstrado a seguir. Note, porém, que ela deve ser filha de `android.support.v4.app.FragmentActivity`. Se você quiser, pode configurar a dependência da biblioteca v7 da action bar de compatibilidade e utilizar a classe `android.support.v7.app.AppCompatActivity`, pois a classe `AppCompatActivity` é filha de `FragmentActivity`. Então, para estes exemplos, tanto faz.

**MainActivity.java**

```
public class MainActivity extends android.support.v4.app.FragmentActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // O FragmentManager é necessário para brincar com os fragments...
        android.support.v4.app.FragmentManager fm = getSupportFragmentManager();
    }
}
```

Para utilizar a biblioteca de compatibilidade v4, configure o arquivo *build.gradle*.

**app/build.gradle**

```
...
dependencies {
    ...
    compile "com.android.support:support-v4:21+"
}
```

Feito isso, crie a classe *Fragment1* conforme demonstrado a seguir.

**Fragment1.java**

```
public class Fragment1 extends android.support.v4.app.Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_1, container, false);
        // O fragment é livre para ter qualquer lógica aqui
        return view;
    }
}
```

Note que essa classe é filha de *android.support.v4.app.Fragment*. Um fragment deve criar e retornar a view no método *onCreateView(inflater, container, bundle)*. A seguir, podemos ver o arquivo de layout que o fragment vai inflar para retornar a view.

**/res/layout/fragment_1.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
```

```

        android:orientation="vertical" android:gravity="center">
        <TextView
            android:id="@+id/text"
            android:layout_width="match_parent" android:layout_height="wrap_content"
            android:gravity="center"
            android:text="Fragment 1" />
    </LinearLayout>

```

Com o fragment criado, basta adicioná-lo no layout da activity, seja de forma estática no XML ou dinamicamente pela API. Para adicionar um fragment no arquivo XML de layout da activity, é utilizada a tag <fragment>, informando a largura e altura do fragment como se fosse uma view. O atributo class recebe o nome completo da classe do fragment.

Nota: Eu particularmente não gosto de utilizar o RelativeLayout. Veja que nos exemplos do livro você vai encontrar o LinearLayout ou FrameLayout como a raiz do layout.

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent" android:layout_height="match_parent" >
    <fragment android:id="@+id/frag1"
        android:layout_width="match_parent" android:layout_height="match_parent"
        class="br.com.livroandroid.cap09_hellofragments.Fragment1"
        tools:layout="@layout/fragment_1" />
</LinearLayout>

```

Atenção: cuidado ao digitar o nome da classe do fragment no arquivo XML da activity. Se você errar o nome, a aplicação vai lançar um erro em tempo de execução. Note que o Android Studio inclusive ajuda a completar o nome da classe para evitar erros. Outra dica é, sempre depois de digitar o nome da classe, segure a tecla Ctrl e clique com o mouse para abrir o arquivo. Se o arquivo da classe abrir, está tudo ok.

A figura 8.9 mostra a pré-visualização do arquivo da activity. O segredo da pré-visualização do layout é o atributo tools:layout="@layout/fragment_1", que é utilizado somente pelo editor visual e faz com que o layout do fragment seja inserido neste local, apenas para fazer a pré-visualização.

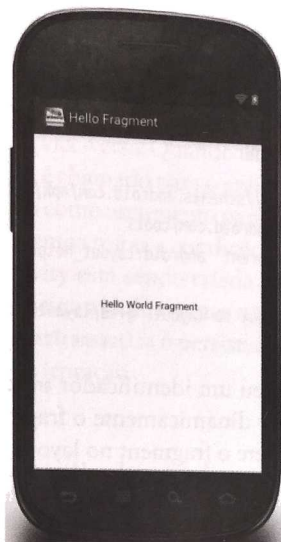


Figura 8.9 – Pré-visualização.

Pronto! Se você executar o projeto no emulador, deve ver a mensagem **Hello World Fragment** na tela. Para fechar esse tópico, veja que o fragment tem um identificador que foi declarado no layout.

```
<fragment android:id="@+id/frag1" ...
```

Isso significa que em qualquer local do código podemos recuperar esse fragment com o método `findFragmentById(id)` da classe `android.support.v4.app.FragmentManager`. O código a seguir demonstra como encontrar um fragment pelo id.



MainActivity.java

```
public class MainActivity extends android.support.v4.app.FragmentActivity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        android.support.v4.app.FragmentManager fm = getSupportFragmentManager();  
        Fragment1 frag1 = (Fragment1) fm.findFragmentById(R.id.frag1);  
        frag1.vocePodeChamarOMetodoQuePrecisarAqui();  
    }  
}
```

Como último exemplo, vamos demonstrar como utilizar a API para adicionar um fragment dinamicamente no layout. Nesse caso, remova a tag <fragment> do layout da activity e deixe o layout vazio, conforme demonstrado a seguir.

```

 /res/layout/activity_main.xml

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:id="@+id/layoutFrag">
    <!-- Fragment será inserido aqui no layout "@+id/layoutFrag" -->
</FrameLayout>

```

Veja que o LinearLayout recebeu um identificador android:id="@+id/layoutFrag", que será utilizado para adicionar dinamicamente o fragment nesse layout pela API. O código da activity que insere o fragment no layout pode ser visto a seguir.

```

 MainActivity.java

public class MainActivity extends android.support.v4.app.FragmentActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Adiciona o fragment dinamicamente pela API
        if(savedInstanceState == null) {
            android.support.v4.app.FragmentManager fm = getSupportFragmentManager();
            FragmentTransaction ft = fm.beginTransaction();
            Fragment1 frag1 = new Fragment1();
            ft.add(R.id.layoutFrag, frag1, "Fragment1");
            ft.commit();
        }
    }
}

```

Para inserir, substituir ou remover um fragment pela API é utilizada a classe FragmentTransaction, e o método commit() efetiva as alterações. O método add(layout, frag, tag) recebe como parâmetro o identificador do layout de onde o fragment deve ser inserido, a instância do objeto do fragment e uma string que é a tag. A tag pode ser utilizada posteriormente para encontrar o fragment com o método findFragmentByTag(tag).

```

Fragment1 frag1 = (Fragment1) fm.findFragmentByTag("Fragment1");

```

Repare que, antes de inserir o fragment pela API, foi validada a condição `if(savedInstanceState == null)`, para ter certeza de que a activity estava sendo criada nesse instante. Se o Bundle não estiver nulo significa que a activity foi destruída e criada novamente. Isso pode acontecer ao girar o dispositivo trocando a orientação de vertical para horizontal, ou vice-versa. Quando uma activity é destruída, o método `onSaveInstanceState(bundle)` é chamado para o aplicativo salvar o estado da tela. Esse mesmo Bundle é passado como argumento para o método `onCreate(bundle)` ao recriar a activity. Por isso, devemos testar a condição `if(savedInstanceState == null)` para ter certeza de que a activity está sendo criada pela primeira vez, pois caso contrário o fragment seria adicionado duas vezes no layout. Portanto, lembre-se: a transação criada pelo `FragmentManager` é persistida durante o ciclo de vida da activity e qualquer troca de orientação.

8.5 Utilizando fragments com action bar + tabs

No capítulo 5, sobre action bar, fizemos um exemplo que mostrou como criar as tabs. Agora vamos criar outro exemplo que vai utilizar action bar + tabs + fragments. Embora as tabs com action bar estejam deprecated (depois falamos mais sobre isso), aprender a utilizá-las é muito importante para o seu aprendizado.

No Android, não importa se você utiliza as tabs ou o menu lateral (Navigation Drawer) como a navegação top-level do seu aplicativo, sempre que você selecionar alguma tab ou opção do menu, o conteúdo da tela precisa ser atualizado sem trocar de activity. E isso é feito com fragments.

O próximo exemplo que vamos estudar é o projeto `Fragments-ActionBarTabs`, disponível nos exemplos deste capítulo. Neste projeto, foi configurada a action bar com três tabs, e ao clicar numa tab um fragment será substituído por outro dentro do layout.

Nota: lembre-se de que a classe `android.support.v7.app.AppCompatActivity` é filha de `android.support.v4.app.FragmentActivity`; por isso, podemos utilizar os fragments e a action bar de compatibilidade.



MainActivity.java

```
public class MainActivity extends android.support.v7.app.AppCompatActivity {  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
    }  
}
```



```

ActionBar actionBar = getSupportActionBar();
actionBar.setNavigationMode(android.app.ActionBar.NAVIGATION_MODE_TABS);
// Tab 1
ActionBar.Tab tab1 = actionBar.newTab().setText("Frag 1");
tab1.setTabListener(new MyTabListener(this, new Fragment1()));
actionBar.addTab(tab1);
// Tab 2
ActionBar.Tab tab2 = actionBar.newTab().setText("Frag 2");
tab2.setTabListener(new MyTabListener(this, new Fragment2()));
actionBar.addTab(tab2);
// Tab 3
ActionBar.Tab tab3 = actionBar.newTab().setText("Frag 3");
tab3.setTabListener(new MyTabListener(this, new Fragment3()));
actionBar.addTab(tab3);
}
}

```

No código da activity, a classe `MyTabListener` recebe no construtor a instância do fragment que deve ser substituído no layout. Para este exemplo criei as classes `Fragment1`, `Fragment2` e `Fragment3`. No arquivo de layout da activity, mais uma vez vamos deixar apenas o layout de marcação, pois os fragments serão inseridos dinamicamente.

 `/res/layout/activity_main.xml`

```

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:id="@+id/layoutFrag">
    <!-- Fragment será inserido aqui no layout "@+id/layoutFrag" -->
</FrameLayout>

```

As classes `Fragment1`, `Fragment2` e `Fragment3` não serão exibidas aqui para economizar espaço no livro, pois elas são simples como a classe `Fragment1` que estudamos no exemplo anterior. Lembrando que cada fragment tem seu próprio arquivo de layout, por exemplo: `/res/layout_fragment_1`, `/res/layout_fragment_2` e `/res/layout_fragment_3`.

O segredo para utilizar os fragments com as tabs é o `TabListener`. Veja que a classe `MyTabListener` recebe no seu construtor o fragment que ela deve mostrar ao clicar em cada tab. Veja que basta uma única linha de código para executar o comando `FragmentManager.replace(...)` e substituir o fragment ao clicar em alguma tab.



MyTabListener.java

```
import android.content.Context;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentTransaction;
import android.support.v7.app.ActionBar;

public class MyTabListener implements ActionBar.TabListener {
    private Context context;
    private Fragment frag;
    public MyTabListener(Context context, Fragment frag) {
        this.context = context;
        this.frag = frag;
    }
    @Override
    public void onTabSelected(ActionBar.Tab tab, FragmentTransaction ft) {
        // Troca o fragment dinamicamente ao clicar na tab
        ft.replace(R.id.layoutFrag, this.frag, null);
    }
    // Métodos onTabUnselected e onTabReselected aqui.
}
```

Ao executar esse projeto, o resultado deve ser como a figura 8.10, que mostra a segunda tab selecionada e o `Fragment2` no layout.

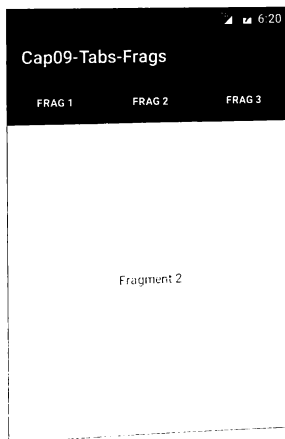


Figura 8.10 – Action bar com tabs + fragments.

8.6 Utilizando fragments com action bar + tabs + ViewPager

Para melhorar ainda mais o exemplo anterior, podemos utilizar o `ViewPager` para controlar os fragments e permitir fazer o gesto de swipe lateral para navegar entre as tabs. Esse é um padrão de design muito conhecido no Android, e você precisa dominá-lo.

O próximo exemplo que vou mostrar é o projeto `Fragments-Tabs-ViewPager`, que está disponível com os exemplos do livro. Neste projeto foi configurado a action bar com três tabs, porém o controle de navegação é feito pelo `ViewPager`. Isso é o mais importante deste exemplo! O `ViewPager` faz todo o trabalho e a tab só mostra a página que está sendo exibida.

O adapter do `ViewPager` será formado pelos três fragments, `Fragment1`, `Fragment2` e `Fragment3`. Portanto, você poderá utilizar o gesto de swipe para navegar nos fragments. A tab nessa história é uma mera coadjuvante, pois ela apenas mostra a página selecionada. A seguir, podemos visualizar o código-fonte da `MainActivity` que demonstra como utilizar o `ViewPager` com fragments.

MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    private ViewPager viewPager;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // ViewPager
        viewPager = (ViewPager) findViewById(R.id.viewPager);
        viewPager.setAdapter(new TabsAdapter(getSupportFragmentManager()));
        // Configura as Tabs
        final ActionBar actionBar = getSupportActionBar();
        actionBar.setNavigationMode(android.app.ActionBar.NAVIGATION_MODE_TABS);
        actionBar.addTab(actionBar.newTab().setText("Frag 1").setTabListener(new
            MyTabListener(viewPager, 0)));
        actionBar.addTab(actionBar.newTab().setText("Frag 2").setTabListener(new
            MyTabListener(viewPager, 1)));
        actionBar.addTab(actionBar.newTab().setText("Frag 3").setTabListener(new
            MyTabListener(viewPager, 2)));
        // Se o ViewPager troca de página, atualiza a Tab.
        viewPager.setOnPageChangeListener(new ViewPager.OnPageChangeListener() {
            @Override
            public void onPageSelected(int idx) {
                // Se fizer swipe no ViewPager, atualiza a tab
                actionBar.setSelectedNavigationItem(idx);
            }
        });
    }
}
```

```

    }
    @Override
    public void onPageScrolled(int position, float positionOffset,
        int positionOffsetPixels) { }
    @Override
    public void onPageScrollStateChanged(int state) { }
  });
}
}

```

Neste código, estamos monitorando o evento de troca de página do `ViewPager`, pois precisamos atualizar o índice da tab selecionada, para corresponder à página que o `ViewPager` está mostrando. O mais importante deste exemplo é você entender que não precisamos atualizar os fragments dinamicamente na tela com a classe `FragmentManager`, pois é o `ViewPager` que controla tudo. No arquivo de layout da activity, basta inserir o `ViewPager`.

 /res/layout/activity_main.xml

```

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent" >
    <android.support.v4.view.ViewPager android:id="@+id/viewPager"
        android:layout_width="match_parent" android:layout_height="wrap_content" />
</FrameLayout>

```

Veja que no código-fonte da activity estamos utilizando a classe `TabsAdapter`, que é o adapter do `ViewPager`. Observe que o `ViewPager` pode utilizar uma implementação de `PagerAdapter` para views normais, ou `FragmentPagerAdapter` quando cada página é representada por um fragment.

 `TabsAdapter.java`

```

import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;

public class TabsAdapter extends FragmentPagerAdapter {
    public TabsAdapter(FragmentManager supportFragmentManager) {
        super(supportFragmentManager);
    }
    @Override
    public int getCount() {
        // O ViewPager vai ter 3 páginas
        return 3;
    }
}

```

```

@Override
public Fragment getItem(int idx) {
    if(idx == 0) {
        return new Fragment1();
    } else if(idx == 1) {
        return new Fragment2();
    }
    return new Fragment3();
}
}

```

Esse adapter apenas fornece o conteúdo do ViewPager; teremos três páginas, e cada uma é um fragment. Para concluir o exemplo, a classe MyTabListener que trata dos eventos das tabs foi alterada, para que, quando selecionar uma tab, a página do ViewPager seja atualizada com o índice da tab selecionada. Lembre-se de que neste exemplo o ViewPager é quem controla tudo, e as Tabs são meras coadjuvantes.

MyTabListener.java

```

import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentTransaction;
import android.support.v4.view.ViewPager;
import android.support.v7.app.ActionBar;

public class MyTabListener implements ActionBar.TabListener {
    private ViewPager viewPager;
    private int idx;
    public MyTabListener(ViewPager viewPager, int idx) {
        this.viewPager = viewPager;
        this.idx = idx;
    }
    @Override
    public void onTabSelected(ActionBar.Tab tab, FragmentTransaction ft) {
        // Navega para a página desejada do ViewPager
        viewPager.setCurrentItem(idx);
    }
    @Override
    public void onTabUnselected(ActionBar.Tab tab, FragmentTransaction ft) {}
    @Override
    public void onTabReselected(ActionBar.Tab tab, FragmentTransaction ft) {}
}

```

Nota: a action bar com tabs foi descontinuada (deprecated); assim, você deve estar vendo alguns alertas (warnings) no código. Mas este exemplo é um clássico que

you need to know, because he also showed how to use the ViewPager + Fragments. The tab here is merely coadjutant and can be substituted by any other component. When the form develops the project of the cars we will use the component TabLayout of the Android Design Support Library.

8.7 Ciclo de vida de um fragment

Um fragment tem um ciclo de vida bem definido, o qual é atrelado ao ciclo de vida da activity. Se você já entendeu como funciona o ciclo de vida de uma activity, como os métodos onCreate(bundle), onStart(), onResume(), onPause(), onStop() e onDestroy(), será bem simples de entender o ciclo de vida de um fragment, pois ele segue o mesmo conceito.

A figura 8.11 relembra os métodos do ciclo de vida de uma activity.

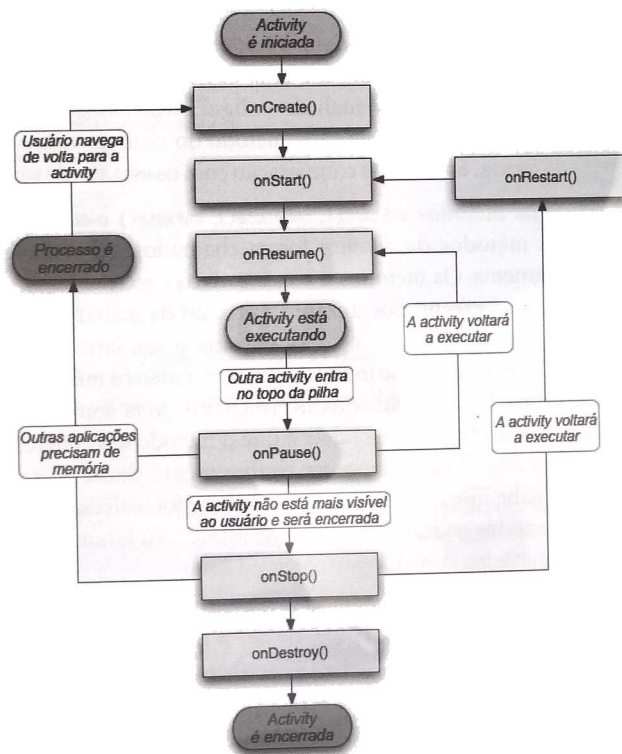


Figura 8.11 – Ciclo de vida de uma activity.

Note que o diagrama está em inglês, pois é da documentação oficial. Caso você precise relembrar em detalhes como funciona o ciclo de vida, verifique o capítulo 4, sobre a classe `Activity`.

O ciclo de vida de uma `activity` contém os tradicionais métodos `onCreate()`, que são chamados uma única vez quando ela é iniciada, e os métodos como `onPause()` e `onStop()`, indicando que a `activity` será interrompida e movida para segundo plano. Outro método clássico é o `onDestroy()`, chamado uma única vez ao destruir a `activity`.

Seguindo esse mesmo princípio, também o ciclo de vida dos `fragments` contém esses mesmos métodos, os quais são amarrados com a `activity` que declarou o `fragment`, conhecida como `host activity`. Portanto, quando algum método do ciclo de vida de uma `activity` for chamado, como por exemplo, o `onPause()`, o sistema vai chamar o método `onPause()` em todos os `fragments` dessa `activity`. Da mesma forma, quando o método `onResume()` da `activity` for chamado, o método `onResume()` de cada `fragment` também será.

A maioria dos métodos do ciclo de vida dos `fragments` é um espelho dos métodos da `activity`, mas também existem outros métodos específicos que existem somente nos `fragments`, conforme podemos visualizar na figura 8.12. No lado esquerdo da figura, podemos ver a ordem em que cada método do ciclo de vida é chamado. No lado direito da figura, é feita uma comparação com os estados de uma `activity`.

Podemos ver que os métodos `onStart()`, `onResume()`, `onPause()` e `onStop()` são simples, e quando os métodos da `activity` forem chamados, eles também serão chamados nos `fragments`. Os métodos `onAttach(activity)`, `onCreate()`, `onCreateView()` e `onActivityCreated()` são executados durante a criação da `activity`, durante o `onCreate()`. Por exemplo, quando uma `activity` informa o seu layout pelo método `setContentView(view)`, os `fragments` são inflados e criados, então os métodos `onAttach()`, `onCreate()` e `onCreateView()` são chamados no `fragment`. Mas somente quando a `activity` retornar do método `onCreate()` dela é que o método `onActivityCreated()` do `fragment` será chamado. É importante ter conhecimento disso, pois nesse momento o `fragment` sabe que a inicialização da `activity` foi realizada com sucesso o que significa que todos os `fragments` do layout também foram inicializados e tiveram suas `views` criadas.

Complementando, quando uma `activity` é destruída durante o seu método `onDestroy()`, os métodos `onDestroyView()`, `onDestroy()` e `onDetach()` são chamados em sequência para encerrar os recursos e desassociar o `fragment` da `activity` que está sendo destruída.

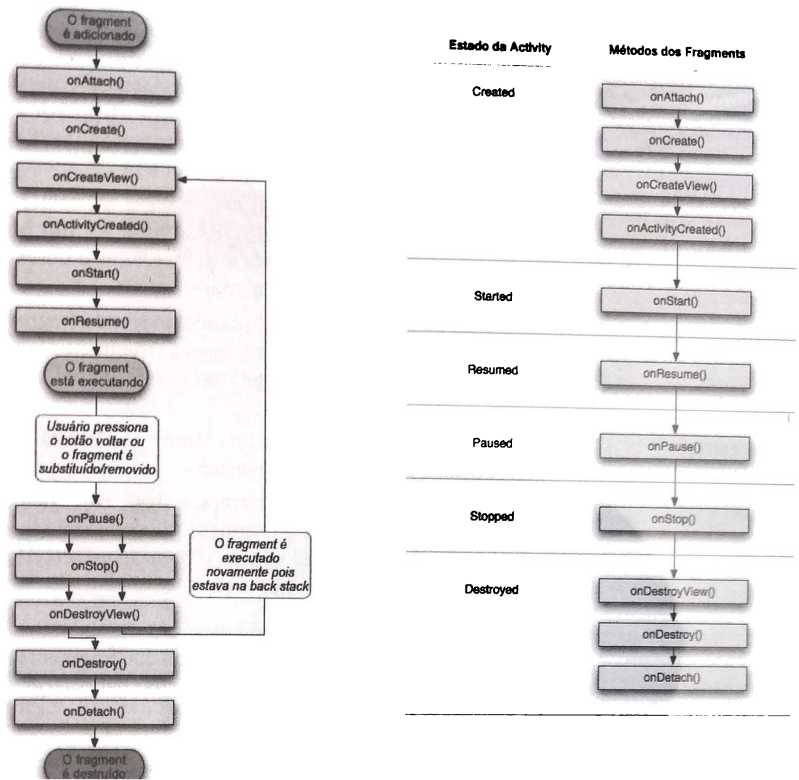


Figura 8.12 – Ciclo de vida de um fragment.

A lista a seguir mostra os principais métodos de ciclo de vida específicos dos fragments:

Método	Descrição
onAttach(activity)	Esse método é chamado logo depois de o fragment ser associado com a activity, o que acontece assim que a activity infla o layout do fragment pela tag <fragment> ou o fragment é adicionado dinamicamente via FragmentTransaction. Note que o importante deste método é que ele recebe como parâmetro a activity que contém o fragment. Somente depois que esse método é chamado (mas não nele), o método getActivity() do fragment vai retornar a activity host.
onCreate(bundle)	Esse método é chamado apenas uma vez e quando o fragment está sendo criado. Ele recebe o Bundle que foi salvo durante o método onSaveInstanceState(state).

Método	Descrição (cont.)
<code>onCreateView(inflater,viewgroup,bundle)</code>	Nesse método, o fragment precisa criar a view que será inserida no layout da activity. Somente depois de esse método retornar, é possível chamar o método <code>getView()</code> que retorna a view que o fragment tem.
<code>onActivityCreated(bundle)</code>	Esse método é chamado logo após o <code>onCreate()</code> da activity ter sido finalizado. Esse pode ser um bom momento para consultar os web services e buscar o conteúdo necessário para criar o layout.
<code>onDestroyView()</code>	Esse método é chamado quando a view do fragment foi removida e não pertence mais ao fragment. Depois desse evento, o método <code>getView()</code> do fragment vai retornar null.
<code>onDestroy()</code>	Chamado para indicar que o fragment não está mais sendo utilizado e será destruído.
<code>onDetach()</code>	Oposto do método <code>onAttach(activity)</code> , esse método é chamado quando o fragment foi desassociado da activity. Depois desse evento, o método <code>getActivity()</code> do fragment vai retornar null.

Dica: não se preocupe se não entender tudo sobre o ciclo de vida dos fragments ou da activity. Continue lendo, e revise esse assunto quando achar necessário.

8.8 Migrando um projeto que utiliza activity para fragments

Um dos exemplos mais comuns de uso dos fragments é um dos primeiros que você precisa entender é como reaproveitar o código entre a versão smartphone e tablet, conforme a figura 8.13.

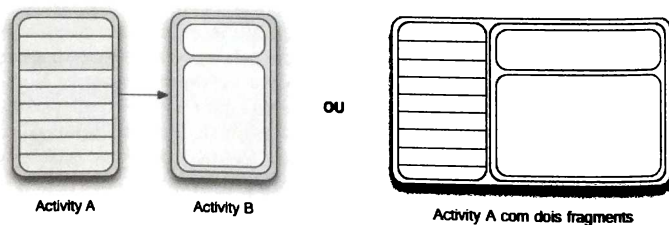


Figura 8.13 – Fragment que divide a tela em pedaços.

No próximo exemplo, vamos criar uma lista com os nomes dos planetas e ao selecionar um item da lista vamos navegar para outra activity, que vai mostrar os detalhes do planeta. O layout será extremamente simples, pois estamos interessados apenas em estudar os fragments. A figura 8.14 mostra o exemplo funcionando. Observe que na segunda tela o nome do planeta é mostrado no título da action bar.

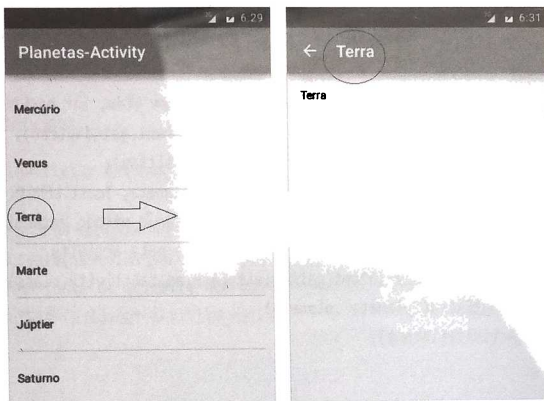


Figura 8.14 – Navegação de telas com duas activities.

Para começarmos o exercício, abra o projeto Planetas-Activity. O código-fonte é simples. Existe uma activity com um ListView na primeira tela e uma activity com apenas um TextView na segunda tela. O exemplo já está funcionando, mas não utiliza fragments. A classe MainActivity é a activity que mostra a lista de planetas, e a classe PlanetaActivity é a segunda tela que recebe o nome do planeta por parâmetro. Por favor, abra o projeto de exemplo no Android Studio e execute no emulador. Dê uma rápida olhada no código para continuarmos o exercício, pois vamos migrar o projeto para utilizar fragments.

A alteração que vamos fazer é encapsular a view e lógica dessas duas activities em fragments. Para começar, vamos criar o fragment PlanetasFragment com o layout do ListView. Basicamente, a lógica da classe MainActivity será transferida para esse fragment.



PlanetasFragment.java

```
public class PlanetasFragment extends android.support.v4.app.Fragment{
    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
        @Nullable Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_planetas, container, false);
```

```

// ListView
ListView listView = (ListView) view.findViewById(R.id.listView);
listView.setAdapter(new PlanetaAdapter(getActivity()));
listView.setOnItemClickListener(onItemClickPlaneta());
return view;
}
private AdapterView.OnItemClickListener onItemClickPlaneta() {
    return new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
            PlanetaAdapter adapter = (PlanetaAdapter) parent.getAdapter();
            String planeta = (String) adapter.getItem(position);
            Toast.makeText(getActivity(), "Planeta: " + planeta, Toast.LENGTH_SHORT).show();
            // O Context é a activity, então pode utilizar o método getActivity()
            // A navegação de telas continua sendo feita pela activity
            Intent intent = new Intent(getActivity(), PlanetaActivity.class);
            intent.putExtra("planeta", planeta);
            startActivity(intent);
        }
    };
}
}
}

```

A diferença ao copiar o código da activity para fragment consiste na utilização do context. Na activity, o context é ela mesma, portanto usamos o `this`. No fragment, o contexto é a activity, portanto utilizamos o método `getActivity()`. Por exemplo, a activity utilizava este código:

```
listView.setAdapter(new PlanetaAdapter(this));
```

Mas agora, dentro do fragment, o `this` vira o `getActivity()`:

```
listView.setAdapter(new PlanetaAdapter(getActivity()));
```

Para continuar a migração do código para fragments, o fragment vai encapsular a lista de planetas; portanto, crie o layout do fragment com o `ListView`.



/res/layout/fragment_planetas.xml

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" android:padding="16dp" >
    <ListView android:id="@+id/listview"
        android:layout_width="match_parent" android:layout_height="match_parent" />
</LinearLayout>

```

Nota: para acessar a classe `android.content.Context` dentro do fragment, utilize o método `getActivity()`, pois a classe `Activity` é filha de `Context`.

O código-fonte do fragment e layout são os mesmos que antes estavam na `MainActivity`. Esse é o papel de um fragment. Ele encapsula determinadas view e lógica da tela. A vantagem disso é que podemos remover a lógica da classe da `activity`, e no layout basta incluir o fragment conforme demonstrado a seguir:



MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // O código-fonte da activity fica vazio.
        setContentView(R.layout.activity_main);
    }
}
```



/res/layout/activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <fragment
        android:layout_width="match_parent" android:layout_height="match_parent"
        class="br.com.livroandroid.planetas.PlanetasFragment"
        tools:layout="@layout/fragment_planetas" />
</RelativeLayout>
```

Feita essa alteração, o código-fonte da `activity` ficou vazio, e apenas a tag `<fragment>` foi inserida no layout. Mais uma vez, vou alertá-lo para tomar cuidado ao digitar o nome da classe do fragment, pois o nome é completo e deve conter o pacote no qual a classe foi criada.

Pronto, já terminamos a primeira `activity`, agora vamos para a segunda. Basicamente, vamos fazer a mesma coisa e criar a classe `PlanetaFragment`. Note que usamos “planeta” no singular, e não no plural, como o fragment que acabamos de criar para a lista.

PlanetaFragment.java

```
public class PlanetaFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_planeta, container, false);
        return view;
    }
    public void setPlaneta(String planeta) {
        TextView text = (TextView) getView().findViewById(R.id.text);
        text.setText("Planeta: " + planeta);
    }
}
```

/res/layout/fragment_planeta.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent" android:layout_height="match_parent"
    tools:context="br.livroandroid.livroandroidcap8_planetas.PlanetaFragment" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="center" />
</FrameLayout>
```

Esse é um fragment simples que apenas mostra o nome do planeta no `TextView`. O único trabalho da activity é inserir esse fragment no seu layout, e passar o parâmetro, que é o nome do planeta para o fragment.

PlanetaActivity.java

```
public class PlanetaActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_planeta);
        // Parâmetro enviado pela intent da activity
        String planeta = getIntent().getStringExtra("planeta");
        // Pega o fragment do layout pelo id
        PlanetaFragment f = (PlanetaFragment)
            getSupportFragmentManager().findFragmentById(R.id.PlanetaFragment);
        // Atualiza o conteúdo do fragment
        f.setPlaneta(planeta);
    }
}
```

```

    // Configura o nome do planeta como titulo na action bar
    getSupportActionBar().setTitle(planeta);
}
}

```

 /res/layout/activity_planeta.xml

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="16dp" >
    <fragment android:id="@+id/PlanetaFragment"
        android:layout_width="match_parent" android:layout_height="match_parent"
        class="br.com.livroandroid.planetas.PlanetaFragment"
        tools:layout="@layout/fragment_planeta" />
</RelativeLayout>

```

Veja que o segredo muitas vezes ao utilizar uma activity com fragments é como passar parâmetros da activity para o fragment. Como a navegação de telas é feita no nível da activity e os parâmetros são passados pela intent, a activity precisa ler esses parâmetros e atualizar o fragment.

Depois dessas alterações, o projeto deve continuar funcionando normalmente. Assim, somente prossiga com a leitura caso seu exemplo esteja funcionando. Se precisar, confira o exercício pronto no projeto de exemplo Planetas-Fragments.

Agora vamos brincar um pouco com fragments e implementar esse exercício de outra maneira. No código que fizemos até o momento, tivemos de passar um parâmetro para o fragment, e, como esse fragment está inserido de forma estática no layout XML, é necessário recuperar o fragment pelo id e chamar um método `setPlaneta(planeta)` para atualizar o conteúdo.

Outra abordagem, que eu até prefiro, é adicionar o fragment dinamicamente pela API. Então para começar, altere o layout da activity do planeta e remova a tag `<fragment>`. Vamos deixar apenas um `FrameLayout` com um identificador para adicionar o fragment.

 /res/layout/activity_planeta.xml

```

<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout . . .
    android:id="@+id/LayoutFrag" >
    <!-- O fragment será adicionado aqui -->
</RelativeLayout>

```

O próximo passo é alterar a activity para adicionar o fragment dinamicamente no layout e pelo método `setArguments(bundle)` do fragment é possível passar os parâmetros. Note que a mágica está em passar o mesmo Bundle que a activity recebeu pela intent.



PlanetaActivity.java

```
public class PlanetaActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_planeta);
        // Mostra o nome do planeta como titulo na action bar
        String planeta = getIntent().getStringExtra("planeta");
        getSupportActionBar().setTitle(planeta);
        if(savedInstanceState == null) {
            PlanetaFragment f = new PlanetaFragment();
            f.setArguments(getIntent().getExtras()); // Parâmetros: mesmo Bundle da intent
            FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
            ft.add(R.id.layoutFrag, f, "PlanetaFragment");
            ft.commit();
        }
    }
}
```

Nota: se for necessário passar um parâmetro para o fragment, a única forma de fazê-lo é adicionar o fragment dinamicamente pela API. O construtor do fragment deve ser sempre vazio, e os parâmetros devem ser passados pelo método `setArguments(bundle)`. Um conceito importante sobre o ciclo de vida da activity e fragments: caso a tela do aplicativo seja rotacionada, o Android vai destruir a activity e seus respectivos fragments. Nesse momento, o método `onSaveInstanceState(bundle)` é chamado para dar a chance ao aplicativo de salvar as informações no Bundle. Logo depois, o Android vai recriar as activities e fragments passando nos métodos `onCreate(bundle)` esse mesmo Bundle com os objetos salvos. É importante ter conhecimento de que informações passadas pela intent das activities e argumentos dos fragments sobrevivem a esse ciclo de vida. Portanto, caso um fragment tenha parâmetros, eles sobrevivem durante a destruição e recriação da activity e seus fragments. Faça o teste, gire a tela do seu smartphone e monitore as chamadas dos métodos do ciclo de vida.

A activity está passando o Bundle com os parâmetros pelo método `setArguments(bundle)`. Isso significa que precisamos ler tais parâmetros no fragment. Para ler esses parâmetros ou argumentos, basta chamar o método `getArguments()`, que vai retornar o Bundle. Note que é uma boa prática validar se o método `getArguments()` não vai retornar nulo, pois isso pode acontecer caso os argumentos não sejam enviados ao fragment.



PlanetaFragment.java

```
public class PlanetaFragment extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle
savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_planeta, container, false);
        if(getArguments() != null) {
            String planeta = getArguments().getString("planeta");
            setPlaneta(planeta);
        }
        return view;
    }
    @Override
    public void onActivityCreated(@Nullable Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        if(getArguments() != null) {
            String planeta = getArguments().getString("planeta");
            setPlaneta(planeta);
        }
    }
    public void setPlaneta(String planeta) {
        TextView text = (TextView) getView().findViewById(R.id.text);
        text.setText("Planeta: " + planeta);
    }
}
```

Nota: eu gosto de utilizar o método `onActivityCreated(bundle)` para iniciar a lógica da tela, porque neste momento sabemos que, pelo ciclo de vida, o método `onCreate(bundle)` da activity já terminou. Outra vantagem é que como o método `onCreateView(...)` do fragment foi chamado e criou a view, podemos chamar o método `getView()` para acessar a view principal do fragment. Se você chamar o método `getView()` durante o método `onCreateView(...)`, o resultado será null.

Pronto! Feito isso, o aplicativo deve continuar funcionando, mas dessa vez o fragment do planeta foi inserido pela API. Eu particularmente gosto de inserir todos os fragments pela API, pois fica mais fácil verificar todos eles no código Java.

8.9 Criando um layout dividido em partes nos tablets

Neste tópico, vamos continuar o exemplo dos planetas. O próximo passo é dividir a tela em duas partes no caso dos tablets.

Para criar um layout específico para tablets, podemos utilizar as seguintes pastas:

Pasta	Descrição
<code>/res/layout-large</code>	Para tablets de 7”.
<code>/res/layout-xlarge</code>	Para tablets de 10”.

Por exemplo, se você adicionar o arquivo `/res/layout-xlarge/activity_main.xml` no projeto, quando o aplicativo for instalado no tablet de 10”, esse layout customizado será utilizado pelo Android. Tudo isso de forma automática, apenas pela convenção de nomes. Lembrando também que, a partir do Android 3.2, foram criadas estas outras pastas:

Pasta	Descrição
<code>/res/layout-sw600dp</code>	Para tablets de 7”.
<code>/res/layout-sw720dp</code>	Para tablets de 10”.

Basicamente as duas pastas indicam o tamanho da tela na horizontal; a notação `sw` é de `smallest width` (menor largura). Os tablets de 7” têm pelo menos 600dp de largura e os de 10” têm pelo menos 720dp de largura. Eu particularmente prefiro utilizar a notação antiga `/res/layout-xlarge` para tablets de 10”, pois funciona desde o Android 3.0 e não apenas no Android 3.2.

Outra pasta importante é a `/res/layout-land`, na qual o qualificador `land` é de `landscape` (horizontal); para customizar o layout para horizontal, basta inserir arquivos nessa pasta. Você pode inclusive fazer uma combinação dos identificadores, por exemplo, a pasta `/res/layout-xlarge-land` é utilizada para tablets de 10” apenas na horizontal.

Para brincarmos com esse conceito, crie o arquivo e pasta `/res/layout-xlarge-land/activity_main.xml`, conforme mostra a figura 8.15.

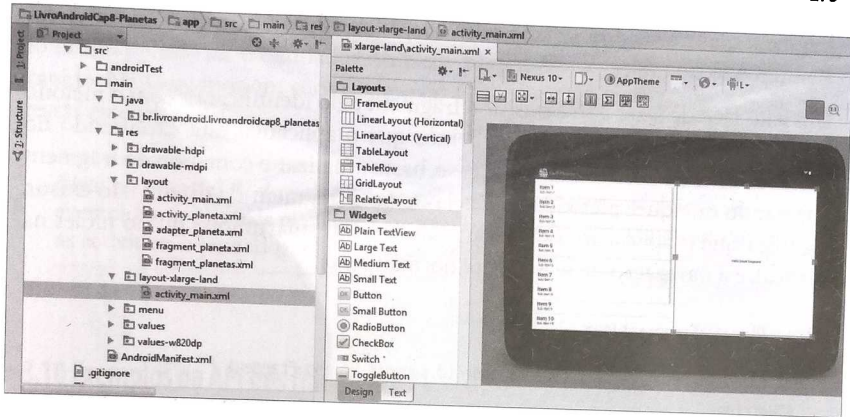


Figura 8.15 – Pré-visualização do layout para tablet.

O código-fonte do arquivo `/res/layout-xlarge-land/activity_main.xml` pode ser visualizado a seguir:

```
res/layout-xlarge-land/activity_main.xml

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
        android:layout_width="match_parent" android:layout_height="match_parent"
        android:orientation="horizontal" >
    <!-- Lista de planetas (esquerda) -->
    <fragment android:id="@+id/PlanetasFragment"
        android:layout_width="0dp" android:layout_weight="1"
        android:layout_height="match_parent"
        class="br.com.livroandroid.planetas.PlanetasFragment"
        tools:layout="@layout/fragment_planetas" />
    <!-- Planeta (direita) -->
    <fragment android:id="@+id/PlanetaFragment"
        android:layout_width="0dp" android:layout_weight="1"
        android:layout_height="match_parent"
        class="br.com.livroandroid.planetas.PlanetaFragment"
        tools:layout="@layout/fragment_planeta" />
</LinearLayout>
```

Depois dessa alteração, ao executar o aplicativo em um tablet de 10" na horizontal, o layout será dividido em duas partes (direita e esquerda). É necessário alterar o código para que, ao selecionar um planeta na lista, ele seja atualizado no

fragment que está na direita; caso contrário, a navegação de activities vai continuar acontecendo no tablet.

Um jeito fácil de fazer isso é buscar o fragment pelo identificador com o método `findFragmentById(id)`. Se ele existir, sabemos que o aplicativo está executando no tablet de 10" e na horizontal. Nesse caso, basta atualizar o conteúdo do fragment chamando qualquer método de sua classe. Se o fragment da direita não existir, significa que o aplicativo está executando em um smartphone ou no tablet na vertical, e a navegação de telas é feita normalmente.

PlanetasFragment.java

```
public class PlanetasFragment extends android.support.v4.app.Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
        @Nullable Bundle savedInstanceState) {
        . . .
    }
    private AdapterView.OnItemClickListener onItemClickPlaneta() {
        return new AdapterView.OnItemClickListener() {
            @Override
            public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
                PlanetaAdapter adapter = (PlanetaAdapter) parent.getAdapter();
                String planeta = (String) adapter.getItem(position);
                Toast.makeText(getActivity(), "Planeta: " + planeta, Toast.LENGTH_SHORT).show();
                PlanetaFragment f = (PlanetaFragment)
                    getFragmentManager().findFragmentById(R.id.PlanetaFragment);
                boolean dualLayout = f != null;
                if(dualLayout) {
                    // Apenas atualiza o fragment na direita se existe
                    f.setPlaneta(planeta);
                } else {
                    // Faz a navegação de telas no caso do smartphone
                    Intent intent = new Intent(getActivity(), PlanetaActivity.class);
                    intent.putExtra("planeta", planeta);
                    startActivity(intent);
                }
            }
        };
    }
}
```

Execute o projeto no emulador com um tablet de 10" e confira o resultado. Gosto muito de utilizar o emulador do Genymotion (genymotion.com), pois é leve e rápido. No Genymotion, você pode criar facilmente o emulador do Nexus 10 e testar esse exemplo.

Dica: no material de download do livro, você vai encontrar os dois projetos de exemplo com a lista de planetas. O primeiro projeto foi construído apenas com as activities; o segundo, com a solução com fragments.

8.10 Exemplos da API dos fragments

Neste tópico, vamos estudar o projeto **DemoFragments**, que demonstra vários recursos dos fragments. Recomendo que você abra esse projeto no Android Studio e siga minhas explicações para entender cada funcionalidade. Os exemplos demonstrados no projeto vão lhe fornecer uma base sólida de conceitos sobre como utilizar a API de fragments.

Não vou explicar o código-fonte inteiro do projeto, pois ele é simples. Vamos apenas às partes principais. A `MainActivity` apresenta várias ações na action bar, que são configuradas no arquivo `/res/menu/menu_main.xml`. A figura 8.16 mostra as opções dos itens da action bar que existem no projeto. Cada opção demonstra uma funcionalidade básica da API de fragments.

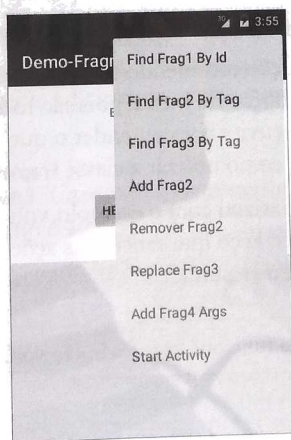


Figura 8.16 – Pré-visualização das ações da action bar.

O arquivo `/res/layout/activity_main.xml` da activity foi dividido em duas partes iguais. Na parte de cima foi inserido o `Fragment1` de forma estática, e a parte de baixo ficou vazia, pois esse será o espaço em que os outros fragments serão inseridos, substituídos ou removidos dinamicamente. A figura 8.17 mostra a pré-visualização do arquivo `/res/layout/activity_main.xml`.



Figura 8.17 – Pré-visualização do layout da activity.

Depois dessa breve introdução, recomendo que você execute esse projeto no emulador e brinque com cada opção do menu, pois são todas simples. Na sequência, estude o código-fonte da activity para entender o que faz cada opção, mas basicamente elas demonstram como utilizar a classe `FragmentManager`.

Uma vez que você se familiarizou com o exemplo, vou dar algumas dicas do que acho importante você saber. Peço que execute as ações exatamente nesta ordem que vou lhe dizer e confira os resultados.

Dica: no emulador, o <ESC> funciona como o botão voltar.

Cenário 1

Clique em **Find Frag2 By Tag**. O resultado é que o fragment não existe.

Clique em **Add Frag2** para adicionar o fragment2 no layout.

Ao clicar em **Find Frag2 By Tag**, o fragment é encontrado.

Clique em **Remover Frag2** para remover o fragment pela API.

Cenário 2

Clique em **Add Frag2**.

Ao clicar em **Find Frag2 By Tag**, o fragment é encontrado.

Clique no botão **Voltar**. Isso vai desfazer a transação do `FragmentManager`, e vai remover o fragment do layout. Isso acontece porque no código foi utilizado o método `addToBackStack(tag)`, que insere a transação na back stack dos fragments.

Clique no botão **Voltar** novamente. Dessa vez, a activity será encerrada e o aplicativo será fechado.

Cenário 3

Clique em **Add Frag2**.

Clique em **Add Frag2** novamente. O frag2 foi adicionado novamente por cima do primeiro.

Clique no botão **Voltar**. Isso removeu o segundo frag2 que estava no topo da pilha.

Clique no botão **Voltar** novamente. Isso removeu o primeiro frag2 que foi adicionado.

Esse exemplo demonstra que devemos ter cuidado ao adicionar os fragments, pois eles ficam por cima dos outros. Como um `FrameLayout` foi utilizado, os fragments adicionados ficam por cima.

Cenário 4

Clique em **Replace Frag3**. Caso não exista nenhum fragment no layout, o `replace` funciona como o `add`.

Clique no botão **Voltar** para desfazer a back stack. O resultado será o layout vazio.

Cenário 5

Clique em **Add Frag2**.

Clique em **Replace Frag3**. O frag2 é substituído pelo frag3.

Clique no botão **Voltar**. A operação é desfeita e o frag2 volta a ocupar o layout.

Clique no botão **Voltar**. O `fragment2` é removido do layout.

Esse exemplo demonstra como funciona a back stack dos fragments. Lembre-se de que isso acontece porque o método `addToBackStack(tag)` foi chamado no código.

Cenário 6

Clique em **Add Frag4 Args**.

Esse exemplo demonstra de forma simples como passar argumentos para o fragment pelo método `setArguments(bundle)`.

Clique no botão **Voltar** para remover o fragment.

Cenário 7

Clique em **Start Activity**.

Esse exemplo demonstra de forma simples como iniciar uma activity que tem um fragment no seu layout. E ainda como passar parâmetros para o fragment.

Clique no botão **Voltar** para voltar à tela anterior.

Cenário 8

Clique no botão **Replace Frag3**.

Note que apareceu uma ação na action bar, com a figura de uma carinha/smile.

Isso significa que fragments podem adicionar ações na action bar.

Ao clicar na ação, o próprio fragment faz o tratamento do evento.

Nos próximos tópicos, vamos revisar alguns conceitos importantes que vimos nos exemplos desse projeto, como a back stack e como os fragments adicionam ações na action bar.

8.11 Back stack

O método `addToBackStack(tag)` da classe `FragmentTransaction` adiciona um fragment na back stack, o que significa que, ao clicar no botão voltar, a operação é desfeita.

O parâmetro do método `addToBackStack(tag)` é a tag que identifica esta transação, mas caso isso não seja importante para a lógica do seu aplicativo é possível passar `null`. O método `popBackStack()` da classe `FragmentManager` é utilizado para desfazer a última operação, ou seja, desempilhar a back stack dos fragments, e funciona como o botão Voltar. O método `popBackStack(string tag, int flags)` da classe `FragmentManager`

recebe a tag que identifica a operação e um flag, que pode ser 0 ou a constante `FragmentManager.POP_BACK_STACK_INCLUSIVE`. Isso faz com que a back stack volte até a tag desejada, por isso você pode marcar a back stack ao chamar o método `addToBackStack(tag)`. Se passar o número 0 (zero) no parâmetro `flags`, a operação é desfeita até a tag informada, mas, se passar `POP_BACK_STACK_INCLUSIVE`, a própria operação da tag informada é desfeita, ou seja, todas as opções são desempilhadas da back stack.

Para concluir a ideia, cada vez que utilizamos a `FragmentTransaction` e fazemos `commit()`, podemos dar um nome para essa transação. É exatamente isso o que faz o método `addToBackStack(tag)`. Ao adicionar esta transação na back stack, tudo pode ser desfeito para voltar ao estado anterior. O método `popBackStack(string tag, int flags)` força o estado a voltar exatamente para o nome da tag que você salvou.

Nota: da mesma forma que o botão voltar é utilizado para desempilhar as activities da pilha de atividades (activity stack), o comportamento é o mesmo caso os fragments sejam adicionados na back stack. Nem sempre você vai utilizar esse recurso, vai depender do que sua aplicação está fazendo, mas é importante conhecer seu funcionamento.

8.12 Adicionando botões na action bar pelo fragment

Um fragment pode adicionar botões na action bar normalmente, mas para isso é obrigatório que ele informe ao sistema por meio da chamada do método `setHasOptionsMenu(true)`.

Se o fragment informar ao sistema que ele deseja adicionar ações na action bar, o método `onCreateOptionsMenu(menu, inflater)` será chamado. Basta inflar o XML de menu. A seguir, podemos visualizar o código-fonte da classe `Fragment3` do exemplo anterior.



Fragment3.java

```
public class Fragment3 extends android.support.v4.app.Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_3, container, false);
        setHasOptionsMenu(true);
        return view;
    }
    @Override
    public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
```



```

        inflater.inflate(R.menu.menu_frag3, menu);
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();
        if (id == R.id.action_hello_frag3) {
            Toast.makeText(getActivity(), "Hello ActionBar Frag 3", Toast.LENGTH_SHORT).show();
            return true;
        }
        return super.onOptionsItemSelected(item);
    }
    public void hello() {
        Toast.makeText(getActivity(), "Hello Frag 3", Toast.LENGTH_SHORT).show();
    }
}

```

 /res/menu/menu_fragment3.java

```

<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto" >
    <item android:id="@+id/action_hello_frag3"
        android:title="@string/hello_frag3" android:icon="@drawable/smile1"
        app:showAsAction="always" />
</menu>

```

A figura 8.18 mostra o botão com o ícone de smile, o qual foi adicionado pelo fragment.

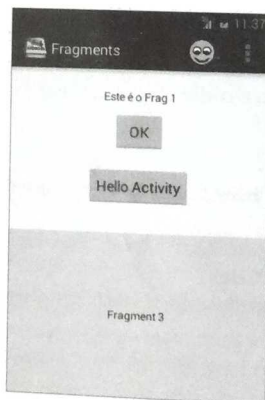


Figura 8.18 – Fragment com ação na action bar.

8.13 Salvando o estado de um fragment

A maioria dos aplicativos consulta dados na internet por meio de um web service, e essa tarefa pode demorar alguns segundos dependendo da velocidade da conexão. Caso o usuário gire a tela, por padrão o Android vai destruir a activity e seus fragments, para depois recriá-los. Nesse momento podemos salvar o estado da tela, para depois recuperar os dados do web service e evitar a necessidade de buscar os dados novamente da internet.

No exemplo anterior, o fragment 1 que foi inserido de forma estática no layout da activity demonstra como salvar o estado. Se você clicar no botão **OK** da tela, verá que uma variável `int count` é incrementada e impressa na tela com um toast. A seguir, podemos visualizar o código simplificado desse fragment, no qual podemos ver como salvar o estado da variável `count`, a fim de preservar o estado durante a troca de orientação da tela.



Fragment1.java

```
public class Fragment1 extends android.support.v4.app.Fragment {
    private int count;
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_1, container, false);
        // Recupera o estado da variável
        if(savedInstanceState != null) {
            count = savedInstanceState.getInt("count");
        }
        view.findViewById(R.id.btOk).setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                count++;
                Toast.makeText(getActivity(), "Count: " + count, Toast.LENGTH_LONG).show();
            }
        });
        return view;
    }
    @Override
    public void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        outState.putInt("count", count); // Salva o estado
    }
}
```

Como vimos, para salvar o estado, basta preencher o Bundle no método `onSaveInstanceState(bundle)`, que é chamado antes de destruir o fragment. Depois, ao recriar a view do fragment, basta ler os dados desse Bundle. Dependendo dos objetos e variáveis do fragment, podemos decidir se é necessário inicializar a tela ou não. Por exemplo, podemos decidir se é necessário buscar os dados do web service ou não.

Outra maneira de preservar o estado do fragment, que é muito utilizada devido sua facilidade, é o método `setRetainInstance(true)`. Caso esse método seja chamado, a instância do fragment será preservada durante a rotação. Assim, o objetivo inteiro do fragment será mantido em memória, e apenas o método `onCreateView(inflater, container, bundle)` será chamado para recriar e retornar a view. Todos os atributos, objetos e variáveis do fragment serão mantidos vivos, pois a instância do objeto do fragment é preservada. Então faça o teste! Basta chamar o método `setRetainInstance(true)` durante os métodos `onCreate()` ou `onCreateView()` do fragment e pronto. Não é necessário implementar o método `onSaveInstanceState(bundle)`.



Fragment1.java

```
public class Fragment1 extends android.support.v4.app.Fragment {
    private int count;
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_1, container, false);
        // Indica que este fragment deve preservar seu estado
        setRetainInstance(true);
        view.findViewById(R.id.btOk).setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                count++;
                Toast.makeText(getActivity(), "Count: " + count, Toast.LENGTH_LONG).show();
            }
        });
        return view;
    }
}
```

Dica: para testar a troca de orientação da tela no emulador, pressione as teclas **Ctrl+F11**.

8.14 Vantagens de utilizar os fragments

Depois de ler este capítulo, vamos revisar alguns conceitos importantes e as vantagens de utilizar fragments.

1. Deixa o código da activity simples, pois a lógica fica nos fragments.
2. Um fragment tem seu próprio ciclo de vida, o qual é associado ao ciclo de vida da activity. Por exemplo, se a activity chamar os métodos `onPause()` ou `onResume()`, eles também serão chamados em todos os fragments da tela para permitir que cada classe gerencie o seu estado.
3. Permite criar componentes reutilizáveis.
4. Permite reutilizar o código entre as versões smartphone e tablet, ou até mesmo reutilizar código dentro do smartphone.
5. Fragments são como miniactivities, pois têm um ciclo de vida bem definido.
6. Fragments são como views, e basta incluí-los no layout.
7. Fragments são utilizados em aplicativos que utilizam action bar com Tabs e `ViewPager`, para gerenciar o conteúdo de cada tab ou página.
8. Com o uso da API, os fragments podem ser adicionados, substituídos e removidos facilmente de qualquer posição do layout.

8.15 Links úteis

Neste capítulo, estudamos uma das principais APIs de desenvolvimento de UI no Android, utilizada principalmente para criar componentes reutilizáveis de código, facilitando a manutenção do projeto.

No projeto dos carros que vamos desenvolver nos próximos capítulos, usaremos fragments em conjunto com a navegação por tabs da action bar. Fique tranquilo, pois ainda vamos estudar muito esse assunto, e o melhor: você vai aprender desenvolvendo o aplicativo dos carros passo a passo.

Separei alguns dos principais links cuja leitura recomendo:

- **Android API Guides – Fragments**
<http://developer.android.com/guide/components/fragments.html>
- **Android API Guides – Supporting Tablets and Handsets**
<http://developer.android.com/guide/practices/tablets-and-handsets.html>
- **Android Training – Building a Dynamic UI with Fragments**
<http://developer.android.com/training/basics/fragments/index.html>



CAPÍTULO 9

Animações

Animações dão um acabamento profissional ao aplicativo, e costumam enriquecer a experiência do usuário.

Neste capítulo, vamos estudar diversos recursos de animações, como movimentar a view pela tela, criar efeitos de transparência e muitos mais.

Vamos estudar a API de Animations, utilizada desde as primeiras versões do Android, assim como a nova API Animator, criada no Android 3.0 Honeycomb. Além disso, veremos diversas dicas sobre animações em geral.

9.1 Drawable Animation

Para começar a brincadeira, vamos estudar a classe `android.graphics.drawable.AnimationDrawable`, utilizada para criar uma animação de figuras como se fosse um GIF animado.

Esse tipo de animação é feito com um arquivo XML na pasta `/res/drawable`, basta configurar a lista de figuras.

 `/res/drawable/list_loading.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<animation-list xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="false" >
    <item android:drawable="@drawable/loading_01" android:duration="200"/>
    <item android:drawable="@drawable/loading_02" android:duration="200"/>
    <item android:drawable="@drawable/loading_03" android:duration="200"/>
    <item android:drawable="@drawable/loading_04" android:duration="200"/>
</animation-list>
```

O arquivo representa uma figura que vai animar as imagens `loading_01`, `loading_02`, `loading_03`, `loading_04` sequencialmente, com um intervalo de 200 milissegundos.

Para ter essa animação em um `ImageView` basta utilizar a propriedade `android:src` normalmente, informando a imagem que contém a lista.

```
<ImageView android:id="@+id/img"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:src="@drawable/list_loading" />
```

No código, podemos usar o método `getDrawable()` para obter o objeto `AnimationDrawable` da animação e chamar o método `start()`.

```
ImageView img = (ImageView) findViewById(R.id.img);
AnimationDrawable anim = (AnimationDrawable) img.getDrawable();
anim.start();
```

Também é possível utilizar a propriedade `android:background` em vez de `android:src`.

```
<ImageView android:id="@+id/img"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:background="@drawable/list_loading" />
```

Neste caso, usaremos o método `getBackground()` para recuperar o `AnimationDrawable`.

```
ImageView img = (ImageView) findViewById(R.id.img);
Animatable anim = (AnimationDrawable) img.getBackground();
anim.start();
```

9.2 Classe Animation

Até antes do Android 3.0, a classe `android.view.animation.Animation` junto com a classe `android.view.animation.AnimationUtils` formaram a dupla principal para trabalhar com animações no Android. A classe `Animation` tem cinco subclasses:

- `AlphaAnimation` – Animação de transparência.
- `RotateAnimation` – Animação de rotação.
- `ScaleAnimation` – Animação de escala.
- `TranslateAnimation` – Animação de movimento.
- `AnimationSet` – Classe utilizada para criar um grupo de animações para serem executadas ao mesmo tempo.

Essas classes de animações podem ser encapsuladas inclusive em arquivos XML, para evitar que o desenvolvedor escreva muito código. Por exemplo, se precisarmos criar uma animação que vai deixar uma view transparente gradativamente, basta este simples trecho de código:

```
View view = ?
Animation a = AnimationUtils.loadAnimation(this, android.R.anim.fade_in);
a.setDuration(2000);
view.startAnimation(a);
```

O método utiliza a classe `AnimationUtils` para carregar uma animação de transparência que vai deixar a view invisível. Note que o parâmetro passado é um recurso de animação que pode ser acessado pela famosa classe `R`. Nesse caso, usamos a classe `android.R` nativa do Android para acessarmos um recurso de animação já existente na plataforma. Também é possível criar animações personalizadas e inseri-las na pasta `/res/anim`. Observe que a classe `AnimationUtils` serve para inflar o arquivo de animação em um objeto do tipo `Animation`. Nesse exemplo estamos carregando a animação `android.R.anim.fade_in`, e isso retorna um objeto do tipo `AlphaAnimation`.

Depois que uma animação é criada, podemos customizar a forma como ela vai ser executada. Nesse exemplo, foi chamado o método `setDuration(millis)` para configurar o tempo em milissegundos que o efeito da animação deve durar. A seguir temos alguns dos principais métodos da classe `Animation`.

Método	Descrição
<code>setDuration(millis)</code>	Configura o tempo em milissegundos que o efeito da animação deve durar.
<code>setFillAfter(boolean)</code>	Este flag indica que o efeito da animação, conhecido como transformação, deve ser mantido ao terminar a execução. O valor default é <code>false</code> . Por exemplo, se criarmos uma animação de <code>fade_out</code> em uma view, a mesma irá desaparecer aos poucos. Mas ao final da animação a view vai voltar ao normal, aparecendo novamente. Para que a view permaneça invisível ao término da animação, esse método deve ser chamado.
<code>setInterpolator(i)</code>	Informa qual a implementação da interface <code>Interpolator</code> que será utilizada. O padrão é <code>LinearInterpolator</code> .
<code>setRepeatCount(int)</code>	Quantidade de repetições que a animação deve efetuar. O padrão é 0 (zero).
<code>setRepeatMode(int tipo)</code>	Configura se a animação deve repetir ou não. É chamado para deixar uma animação em loop: <code>setRepeatMode(Animation.INFINITE)</code> .

Por último, depois que a animação foi criada, basta chamar o método `view.startAnimation(animation)` para iniciar a animação.

Nota: é recomendável criar as animações utilizando arquivos de XML dentro da pasta `/res/anim`. Podemos encontrar várias animações já disponíveis de forma nativa no Android, dispensando a criação de arquivos de animação

customizados. Por exemplo, as animações `fade_in` e `fade_out` já estão presentes no Android; basta selecionar a classe `android.R` para utilizar os recursos nativos. Dentro da pasta `/android-sdk/platforms/${plataforma}/data/res/anim` você pode encontrar as animações padrões da plataforma.

9.3 View Animation

View Animation é o framework de animações até o Android 2.x, e com ele podemos criar efeitos de transparência, movimento, escala e rotação de forma simples.

Nos próximos tópicos, vou explicar como criar cada um desses tipos de animação, e recomendo que você abra o projeto `ViewAnimation` de exemplo deste capítulo para acompanhar as explicações. É importante que você execute o código no emulador para visualizar as animações acontecendo. A figura 9.1 mostra o projeto de exemplo executando no emulador, com um exemplo da animação de transparência, que é a classe `AlphaAnimation`, a qual vamos estudar no próximo tópico.

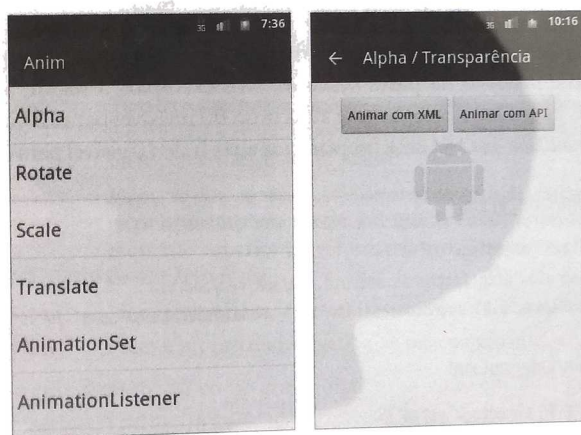


Figura 9.1 – Imagem quase desaparecendo durante a animação de transparência.

9.4 AlphaAnimation

A classe `AlphaAnimation` permite criar os famosos efeitos de `fade_in` e `fade_out`, para alterar a transparência da view, com o objetivo de mostrar ou esconder a view.

Na prática, a classe `AlphaAnimation` altera a propriedade `alpha` da `view`, responsável pela transparência. Sempre que o `alpha` for 0.0 (zero), significa que a `view` está invisível, e sempre que o `alpha` for 1.0 ela está 100% visível. A propriedade `alpha` é um `float` que varia entre 0.0 e 1.0. A tabela a seguir resume os dois parâmetros utilizados pela classe `AlphaAnimation`.

Propriedade	Descrição
<code>float fromAlpha</code>	Valor inicial da propriedade <code>alpha</code> .
<code>float toAlpha</code>	Valor final da propriedade <code>alpha</code> .

Para usar a classe `AlphaAnimation`, podemos utilizar um trecho de código como este:

```
View view = /* qualquer view */
boolean show = false; // Mostrar ou não a view?
AlphaAnimation fade_out = new AlphaAnimation(1.0f, 0.0f); // Apaga a view (alpha 1 para 0)
AlphaAnimation fade_in = new AlphaAnimation(0.0f, 1.0f); // Mostra a view (alpha 0 para 1)
AlphaAnimation a = show ? fade_in : fade_out;
a.setDuration(2000); // 2 segundos
a.setFillAfter(true); // Manter o efeito no final da animação
view.startAnimation(a);
```

Outra forma de fazer a animação é criar um arquivo XML com a definição da animação e inseri-lo na pasta `/res/anim`. Nos exemplos a seguir, a animação `/res/anim/fade_in.xml` faz a propriedade `alpha` ir do 0.0 invisível para 1.0 visível. A animação `/res/anim/fade_out.xml` faz a propriedade `alpha` ir do 1.0 visível para 0.0 invisível.

 `/res/anim/fade_in.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
    android:fromAlpha="0.0" android:toAlpha="1.0" android:duration="2000" />
```

 `/res/anim/fade_out.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
    android:fromAlpha="1.0" android:toAlpha="0.0" android:duration="2000" />
```

Uma vez que os recursos de animação foram criados, podemos utilizar a classe `AnimationUtils` para carregar e iniciar a animação. Nesse momento podemos utilizar a classe `R` do nosso próprio projeto para acessar esses recursos.

```
View view = /* qualquer view */
boolean show = false; // Mostrar ou não a view?
int anim = show ? R.anim.fade_in : R.anim.fade_out;
```

```
Animation a = AnimationUtils.loadAnimation(this, anim);
view.startAnimation(a);
```

Nota: estou mostrando apenas o trecho de código necessário para criar as animações, por isso recomendo que execute no emulador o projeto de exemplo deste capítulo.

9.5 RotateAnimation

A classe `RotateAnimation` serve para para rotacionar uma `view`, na qual são utilizadas as coordenadas `x` e `y` e um ângulo em graus para a rotação. Para essa animação, é possível configurar um ponto que será o eixo da rotação. O padrão da rotação é a partir do canto superior esquerdo da `view`, que tem os pontos `x = 0` e `y = 0`.

A tabela a seguir resume os dois parâmetros utilizados pela classe `RotateAnimation`:

Propriedade	Descrição
<code>float fromDegrees</code>	Valor inicial do ângulo para rotacionar.
<code>float toDegrees</code>	Valor final do ângulo para rotacionar.
<code>int pivotXType</code>	Tipo da rotação aplicada baseada no eixo <code>x</code> , podendo assumir uma das três constantes: <code>Animation.ABSOLUTE</code> , <code>Animation.RELATIVE_TO_SELF</code> ou <code>Animation.RELATIVE_TO_PARENT</code> .
<code>float pivotXValue</code>	Valor no eixo <code>x</code> para servir de eixo da rotação. Pode receber um valor absoluto em pixels, em que 0 (zero) representa o canto superior esquerdo ou valores relativos à coordenada da <code>view</code> ou de seu <code>layout</code> pai.
<code>int pivotYType</code>	Idem a propriedade <code>pivotXType</code> mas para o eixo <code>y</code> .
<code>float pivotYValue</code>	Idem a propriedade <code>pivotXValue</code> mas para o eixo <code>y</code> .

O código a seguir demonstra como rotacionar uma `view` em 180° graus com a classe `RotateAnimation`. A primeira animação gira a `view`, e a segunda gira novamente para a posição de origem.

```
View view = /* qualquer view */
boolean primeiraVez = true;
int angulo = 180; // Girar 180° graus
Animation giraIda = new RotateAnimation(0, angulo,
    Animation.RELATIVE_TO_SELF, 0.5f,
    Animation.RELATIVE_TO_SELF, 0.5f);
Animation giraRetorno = new RotateAnimation(angulo, 0,
    Animation.RELATIVE_TO_SELF, 0.5f,
```

```

        Animation.RELATIVE_TO_SELF, 0.5F);
        Animation a = primeiraVez ? giraIda : giraRetorno;
        a.setDuration(2000); // 2 segundos
        a.setFillAfter(true); // Manter o efeito no final da animação
        view.startAnimation(a);

```

Note que foram passados no construtor da classe `RotationAnimation` os graus inicial e final da rotação (0° e 180°), e as coordenadas X e Y que são o eixo da rotação. Para a animação de retorno, foram definidos os graus inicial e final da rotação (180° e 0°).

Foi informado `Animation.RELATIVE_TO_SELF` para indicar que a coordenada é relativa ao próprio objeto, sendo que o valor 0.5F corresponde a 50%. Nesse caso, o valor 0.0F é igual a 0%, 0.5F é igual a 50% e 1.0F é igual a 100%. A constante `Animation.RELATIVE_TO_SELF` é utilizada para indicar que a rotação é relativa ao objeto que está sendo animado. Dessa forma, sabemos que o ponto X e Y que indicamos com o valor 0.5F (50%) é o centro da imagem. Se alterarmos esta constante para `Animation.RELATIVE_TO_PARENT` este ponto de eixo passaria a ser o centro do layout. A diferença na animação seria gritante. Ao rotacionar no eixo de si mesma, a imagem nem sequer se move de lugar, ela apenas gira. Mas, se rotacionarmos a view utilizando outro ponto como o centro do layout, ela iria se movimentar ao redor desse ponto.

A seguir podemos ver os dois arquivos XML que fazem esta mesma rotação de girar a view em 180° . Observe que no arquivo o ângulo de rotação vai de 0° a 180° para dar um giro completo, e o eixo da rotação foi definido no centro da imagem, conforme as propriedades `android:pivotX` e `android:pivotY`, que no XML podem receber o valor de 50%.

 /res/anim/rotate_gira_ponta_cabeca.xml

```

<?xml version="1.0" encoding="utf-8"?>
<!-- Rotação do ângulo 0 para 180 graus. -->
<rotate xmlns:android="http://schemas.android.com/apk/res/android"
        android:fromDegrees="0" android:toDegrees="180"
        android:pivotX="50%" android:pivotY="50%"
        android:duration="5000" android:fillAfter="true" />

```

 /res/anim/rotate_gira_ponta_cabeca_retorno.xml

```

<?xml version="1.0" encoding="utf-8"?>
<rotate xmlns:android="http://schemas.android.com/apk/res/android"
        android:fromDegrees="180" android:toDegrees="0"
        android:pivotX="50%" android:pivotY="50%"
        android:duration="5000" android:fillAfter="true" />

```

Utilizando esses XMLs, podemos inflar o objeto da animação:

```
Animation giraIda = AnimationUtils.loadAnimation(this, R.anim.rotate_gira_ponta_cabeca);  
Animation giraRetorno = AnimationUtils.loadAnimation(this,  
    R.anim.rotate_gira_ponta_cabeca_retorno);
```

A figura 9.2 exibe o resultado desse exemplo, e mostra o bonequinho do Android de ponta cabeça depois da rotação.

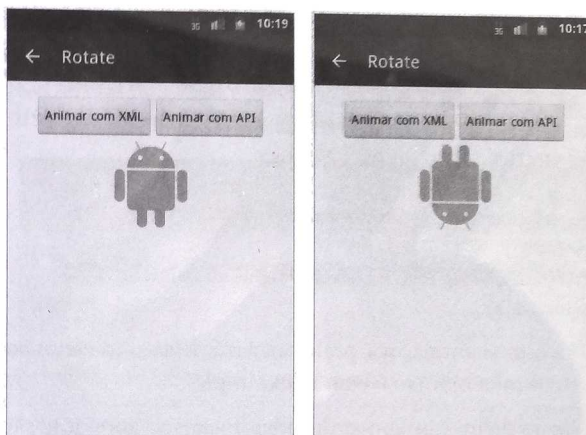


Figura 9.2 – Exemplo da imagem rotacionada em 180°.

Nota: a notação de percentual (exemplo `android:pivotX="100%"`) no XML de animação vai por padrão considerar a configuração relativa à view, como se tivesse utilizado `Animation.RELATIVE_TO_SELF`. Para fazer com que esse parâmetro seja considerado relativo ao layout pai, deve-se inserir um "p" depois do sinal de percentual, por exemplo, `android:pivotX="100%p"`. Dessa forma seria como se `Animation.RELATIVE_TO_PARENT` fosse usado pela API. É interessante que você brinque bastante alterando os parâmetros do eixo da rotação, para que entenda bem o funcionamento das animações.

9.6 ScaleAnimation

A classe `ScaleAnimation` permite aplicar animações de escala, para diminuir ou aumentar uma view. O exemplo de código a seguir faz a view diminuir gradativamente até ela desaparecer. Na próxima vez, vamos fazer com que ela aumente de novo, gradativamente, até o seu tamanho normal.

```

View view = /* qualquer view */
boolean primeiraVez = true; // Se for a primeira vez, vai diminuir...
int angulo = 180;
ScaleAnimation encolher = new ScaleAnimation(
    1.0f, 0.0f, // X inicial e final
    1.0f, 0.0f, // Y inicial e final
    Animation.RELATIVE_TO_SELF, 0.5f, // Eixo X
    Animation.RELATIVE_TO_SELF, 0.5f // Eixo Y
);
ScaleAnimation aumentar = new ScaleAnimation(
    0.0f, 1.0f, // X inicial e final
    0.0f, 1.0f, // Y inicial e final
    Animation.RELATIVE_TO_SELF, 0.5f, // Eixo X
    Animation.RELATIVE_TO_SELF, 0.5f // Eixo Y
);
Animation a = primeiraVez ? encolher : aumentar;
a.setDuration(2000); // 2 segundos
a.setFillAfter(true); // Manter o efeito no final da animação
view.startAnimation(a);

```

Assim como as outras animações, podemos fazer o mesmo efeito com recursos XML, o que em minha opinião é bem mais simples.

O exemplo a seguir define que as coordenadas iniciais da animação são 1.0 (100%), tanto de X quanto de Y, informando o tamanho total da imagem. E as coordenadas finais informadas são 0.0 (0%), informando que o tamanho final da imagem deve ser reduzido em nada. Como o eixo da animação foi definido como 50%, a imagem vai diminuir aos poucos até desaparecer no seu centro.

```

<?xml version="1.0" encoding="utf-8"?>
<scale xmlns:android="http://schemas.android.com/apk/res/android"
    android:fromXScale="1.0" android:toXScale="0.0"
    android:fromYScale="1.0" android:toYScale="0.0"
    android:pivotX="50%" android:pivotY="50%"
    android:fillAfter="true" android:duration="2000"
/>

```

A próxima animação é o contrário da outra. As coordenadas iniciais são 0.0, indicando que a imagem não existe, pois está com seu tamanho zerado. Dessa forma as coordenadas finais foram definidas como 1.0 (100%), para que a imagem volte ao seu tamanho inicial.

 /res/anim/scale_aumentar.xml

```
<?xml version="1.0" encoding="utf-8"?>
<scale xmlns:android="http://schemas.android.com/apk/res/android"
    android:fromXScale="0.0" android:toXScale="1.0"
    android:fromYScale="0.0" android:toYScale="1.0"
    android:pivotX="50%" android:pivotY="50%"
    android:fillAfter="true" android:duration="2000" />
```

A figura 9.3 mostra o resultado desse exemplo. Na primeira animação, a imagem vai diminuir até desaparecer, e depois na próxima vez vai aumentar gradativamente até ficar com o tamanho original. Na parte direita da figura, podemos ver a imagem diminuindo, quase antes de desaparecer.

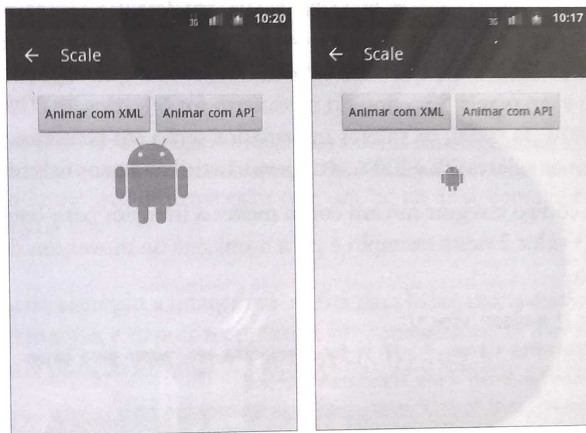


Figura 9.3 – Exemplo da imagem diminuindo de tamanho.

9.7 TranslateAnimation

A classe `TranslateAnimation` serve para mover uma view pelo layout especificando as coordenadas X e Y iniciais e finais da animação. Essas coordenadas podem ser passadas como um valor absoluto `ABSOLUTE` em pixels de tela, ou relativo à própria view `RELATIVE_TO_SELF` ou ao seu layout pai `RELATIVE_TO_PARENT`.

A tabela a seguir resume os parâmetros utilizados pela classe `TranslateAnimation`.

Propriedade	Descrição
<code>float fromXType</code>	Tipo da coordenada x inicial.
<code>float fromXValue</code>	Valor da coordenada x inicial.
<code>int toXType</code>	Tipo da coordenada x final.
<code>float toXValue</code>	Valor da coordenada x final.
<code>int fromYType</code>	Idem explicado no eixo x.
<code>float fromYValue</code>	Idem explicado no eixo x.
<code>int toYType</code>	Idem explicado no eixo x.
<code>float toYValue</code>	Idem explicado no eixo x.

Basicamente, a classe recebe as coordenadas X e Y iniciais e finais. Temos assim quatro pontos. Para cada ponto, é especificado qual o tipo do valor, que pode ser `Animation.ABSOLUTE`, `Animation.RELATIVE_TO_SELF` ou `Animation.RELATIVE_TO_PARENT`. Sempre que o tipo de um valor for `Animation.ABSOLUTE`, aquele parâmetro poderá receber valores absolutos, que são valores numéricos que especificam a real posição x ou y em pixels. Se o tipo do parâmetro for `Animation.RELATIVE_TO_SELF` ou `Animation.RELATIVE_TO_PARENT`, os valores informados serão em percentual. No XML são utilizados os valores 0% e 100%, e no Java são utilizados os valores 0.0F e 1.0F.

O trecho de código a seguir mostra como mover a imagem para baixo e depois para cima. O valor 2 neste exemplo é para a imagem se mover em duas vezes o seu tamanho.

```
View view = /* qualquer view */
boolean primeiraVez = true; // Se for a primeira vez, mover para baixo.
Animation moverParaBaixo = new TranslateAnimation(
    Animation.RELATIVE_TO_SELF, 0.0F, Animation.RELATIVE_TO_SELF, 0.0F,
    Animation.RELATIVE_TO_SELF, 0.0F, Animation.RELATIVE_TO_SELF, 2.0F
);
Animation moverParaCima = new TranslateAnimation(
    Animation.RELATIVE_TO_SELF, 0.0F, Animation.RELATIVE_TO_SELF, 0.0F,
    Animation.RELATIVE_TO_SELF, 2.0F, Animation.RELATIVE_TO_SELF, 0.0F
);
Animation a = primeiraVez ? moverParaBaixo : moverParaCima;
a.setDuration(2000); // 2 segundos
a.setFillAfter(true); // Manter o efeito no final da animação
view.startAnimation(a);
```

Também podemos criar animações de movimento utilizando XML. O código a seguir mostra uma animação que move a imagem para baixo (note que só é utilizado o eixo y). O valor 200% é para a imagem se mover duas vezes o seu tamanho.

 /res/anim/translate_mover_para_baixo.xml

```
<?xml version="1.0" encoding="utf-8"?>
<translate xmlns:android="http://schemas.android.com/apk/res/android"
    android:fromXDelta="0.0" android:toXDelta="0.0"
    android:fromYDelta="0.0" android:toYDelta="200%"
    android:fillAfter="true" android:duration="2000" />
```

E a animação a seguir é o contrário, faz com que a imagem volte para cima.

 /res/anim/translate_mover_para_cima.xml

```
<?xml version="1.0" encoding="utf-8"?>
<translate xmlns:android="http://schemas.android.com/apk/res/android"
    android:fromXDelta="0.0" android:toXDelta="0.0"
    android:fromYDelta="200%" android:toYDelta="0.0"
    android:fillAfter="true" android:duration="2000" />
```

Nota: no arquivo de animação em XML os valores em percentuais são sempre relativos à view que está sendo animada. Se o desejado for informar um valor relativo ao layout pai, utilize o valor com um "p" no final, como por exemplo `android:toYDelta="50%p"`.

Ao executar esse exemplo, a imagem vai mover para baixo numa distância de duas vezes o seu tamanho, e depois na próxima vez a imagem vai se mover para cima até voltar à sua posição original. A figura 94 mostra o resultado dessa animação.

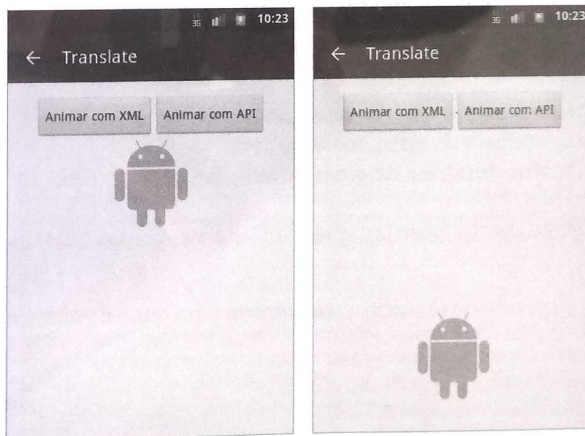


Figura 94 – Exemplo da imagem depois de se mover para baixo.

9.8 AnimationSet

A classe `AnimationSet` permite agrupar várias animações para serem executadas ao mesmo tempo. A maneira de utilizá-la é simples: basta criar um objeto `AnimationSet` e adicionar as animações que devem ser executadas.

A classe `AnimationSet` contém ainda os métodos para configurar as propriedades das animações, como o método `setDuration(long)`, o qual altera o tempo de execução. Se alguma propriedade for alterada nessa classe, ela vai sobrescrever as propriedades que foram antes configuradas em cada animação que foi adicionada na lista. Para facilitar o entendimento, o código-fonte a seguir mostra como mover uma imagem ao mesmo tempo em que ela vai ficando transparente, com a união das classes `TranslateAnimation` e `AlphaAnimation`.

```
AnimationSet lista = new AnimationSet(true);
Animation a1 = getAnimacaoMoverParaBaixoCima();
Animation a2 = getAnimacaoAparecerDesaparecer();
lista.addAnimation(a1);
lista.addAnimation(a2);
// Pronto, basta utilizar a animação agora.
```

Se você preferir, também é possível criar esta lista de animações no arquivo XML, no qual as tags de cada animação são agrupadas dentro de uma tag `<set>`. O exemplo a seguir mostra como criar a animação que move a view para baixo ao mesmo tempo que faz ela desaparecer.

 /res/anim/set_mover_para_baixo_desaparecer.xml

```
<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
    <translate
        android:fromXDelta="0.0" android:toXDelta="0.0"
        android:fromYDelta="0.0" android:toYDelta="200%"
        android:fillAfter="true" android:duration="2000" />
    <alpha
        android:fromAlpha="1.0" android:toAlpha="0.0" android:duration="2000" />
</set>
```

No próximo arquivo é o contrário, a view move para cima e volta a aparecer.

```
 /res/anim/set_mover_para_cima_aparecer.xml

<?xml version="1.0" encoding="utf-8"?>
<set xmlns:android="http://schemas.android.com/apk/res/android">
  <translate
    android:fromXDelta="0.0" android:toXDelta="0.0"
    android:fromYDelta="200%" android:toYDelta="0.0"
    android:fillAfter="true" android:duration="2000" />
  <alpha
    android:fromAlpha="0.0" android:toAlpha="1.0" android:duration="2000" />
</set>
```

Agora podemos carregar o `AnimationSet` utilizando estes recursos de animação:

```
AnimationSet set1 = (AnimationSet) AnimationUtils.loadAnimation(this,R.anim.set_mover_
para_baixo_desaparecer);
AnimationSet set2 = (AnimationSet) AnimationUtils.loadAnimation(this, R.anim.set_mover_
para_cima_aparecer);
AnimationSet set = flag ? set1 : set2;
// Pronto, basta utilizar a animação agora.
```

Lembre-se de que as animações adicionadas no `AnimationSet` serão agrupadas em uma única animação. Os efeitos e transformações de cada animação vão acontecer tudo ao mesmo tempo, o que pode resultar em diferentes animações. É interessante que você crie alguns exemplos para se acostumar com o comportamento das animações. A figura 9.5 exibe o resultado da animação. Na parte direita da figura, podemos ver a imagem se movendo para baixo e quase invisível.

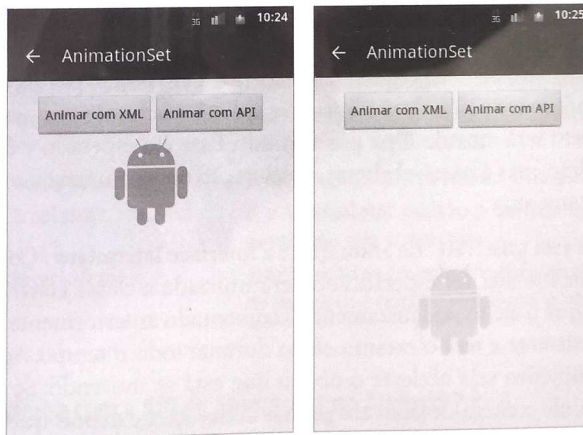


Figura 9.5 – Exemplo de `AnimationSet`.

9.9 AnimationListener

Se quiser monitorar o início e término de uma animação, basta usar a interface `android.view.animation.Animation.AnimationListener`.

Isso é feito chamando o método `setAnimationListener(AnimationListener)` da classe `Animation`, conforme demonstrado a seguir:

```
Animation anim = ...;
anim.setAnimationListener(new AnimationListener() {
    @Override
    public void onAnimationStart(Animation animation) {
        // A animação foi iniciada
    }
    @Override
    public void onAnimationEnd(Animation animation) {
        // A animação terminou
    }
    @Override
    public void onAnimationRepeat(Animation animation) {
        // A animação está repetindo
    }
});
```

9.10 Interpolator

Imagine que você possui uma animação que vai mover o objeto da esquerda para direita com o tempo de duração de dez segundos. De certa forma, podemos dizer que, se dividirmos a distância que será percorrida pelo objeto por 10, o objeto vai a cada segundo mover 10% da distância. Se a distância total a ser percorrida for 100px, o objeto será movido 10px por segundo. Esse é o esperado e é o comportamento padrão, mas é possível alterar a aceleração dessa animação e customizar seu comportamento.

Quem define esta taxa "rate" da animação é a interface `Interpolator`. Como padrão, se nenhum interpolator for especificado, será utilizada a classe `LinearInterpolator`, que faz com que o efeito seja justamente o comentado anteriormente, e a animação será consistente e terá o mesmo efeito durante todo o tempo. Agora vamos dizer que o objetivo seja acelerar o objeto que está se movendo. Se pensarmos em um carro, ele começa devagar até ganhar aceleração, e depois que embala vai embora. Mas lembre-se: muita calma nessa hora. Podemos criar o mesmo efeito

com a animação de um objeto e acelerá-la aos poucos. Usamos para isso a classe `AccelerateInterpolator`. Para informar qual interpolator deve ser selecionado, chame o método `setInterpolator(interpolator)` da classe `Animation`:

```
Animation anim = ...;
anim.setInterpolator(new AccelerateInterpolator());
```

E para definir o interpolator utilizando o XML, utilize o atributo `android:interpolator`:

```
<?xml version="1.0" encoding="utf-8"?>
<translate . . . android:interpolator="@android:anim/accelerate_interpolator" />
```

Existem diversas classes que implementam a interface `Interpolator`. Fica de exercício para você brincar com elas para testar a diferença que cada uma faz nos efeitos de cada animação. Na lista a seguir, podemos verificar a explicação das principais classes.

Método	Descrição
<code>AccelerateDecelerateInterpolator</code>	A animação começa rápido e termina devagar, mas bem no meio dá uma acelerada.
<code>AccelerateInterpolator</code>	A animação começa devagar e vai acelerando.
<code>AnticipateInterpolator</code>	A animação começa para trás e depois vai animando para frente. Esse interpolator dá a impressão de que o objeto vai para trás para pegar um “embalo” antes de acelerar.
<code>AnticipateOvershootInterpolator</code>	Idem à animação <code>AnticipateInterpolator</code> , mas depois de pegar o “embalo” ela se empolga e acaba passando do alvo, de forma que tem de voltar um pouco.
<code>BounceInterpolator</code>	Esse interpolator faz com que o objeto dê uns pulinhos ao atingir o final da animação, como se uma bola estivesse quicando no chão antes de parar.
<code>CycleInterpolator</code>	Faz com que a animação seja repetida no final por um número “n” de vezes.
<code>DecelerateInterpolator</code>	A animação começa rápido e vai desacelerando.
<code>LinearInterpolator</code>	É o interpolator padrão e faz com que o efeito da animação seja constante.
<code>OvershootInterpolator</code>	A animação se empolga e acaba passando do alvo, de forma que tem de voltar um pouco.

9.11 O problema com a API de animações no Android 2.x

No Android 2.x, a API de animação é conhecida como **View Animation**, e com ela é possível criar vários efeitos especiais conforme acabamos de estudar.

Mas esse framework de animação tem um problema clássico que vou explicar agora. O framework da **View Animation** altera apenas a aparência de uma view, mas não o seu conteúdo interno, ou seja, o estado do objeto. Por exemplo, se fizermos uma animação para mover um botão para baixo, o esperado é que o evento seja disparado ao clicar na nova posição do botão. Mas não é isso o que acontece, pois o botão apenas foi desenhado em outro local, mas é como se ele ainda estivesse lá no local original. O usuário teria de continuar clicando no local de origem do botão para disparar o evento.

No projeto de exemplo deste capítulo, você vai encontrar um exemplo que demonstra essa situação, conforme a figura 96. A primeira parte da figura mostra o botão antes da animação e a segunda parte mostra o botão lá em baixo, pois ele se moveu. Conforme já expliquei, você terá de clicar na área definida pelo círculo, mesmo depois de o botão estar em outra posição.

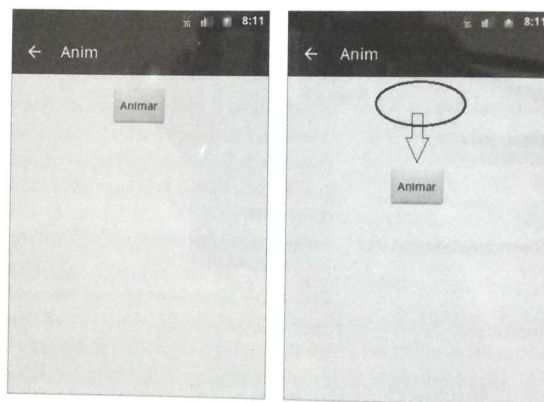


Figura 96 – Animação de movimento.

Nota: View Animation é o framework de animação do Android 2.x, o qual altera apenas a maneira como a view é desenhada, mas mantém as propriedades da view inalteradas, o que pode causar alguns problemas.

9.12 Property Animations

No Android 3.0 (Honeycomb) foi criada a API Property Animation, que consiste em um robusto framework de animações, que pode animar e alterar qualquer propriedade

de um objeto, sobre uma linha de tempo. Essa nova API de animação vem complementar a API de animação do Android 2.x, conhecida como **View Animation**.

Um dos objetivos desse novo framework é solucionar o problema que vimos anteriormente, em que a animação não alterava as propriedades internas do objeto, apenas mudava a forma como o mesmo era desenhado. Com a **Property Animation**, o conceito de animação é baseado nas propriedades do objeto que podem ser alteradas, como por exemplo a propriedade `alpha` e as posições `x` e `y`. Em vez de criar classes de objetos como `AlphaAnimation` e `TranslateAnimation`, podemos executar um código que vai alterar determinadas propriedades sobre uma linha do tempo. Para criar o `AlphaAnimation`, basta alterar a propriedade `alpha` que controla a transparência, e para mover a view basta alterar as propriedades `x` e `y`.

O interessante desse framework é que você pode animar uma série de propriedades, conforme vamos aprender agora.

9.13 Classe ValueAnimator

A primeira classe que vamos estudar no novo framework é a `ValueAnimator`, que consiste em criar uma animação genérica e utilizar um listener para ouvir os resultados durante a animação.

Nos próximos exemplos, usaremos o mesmo layout com a figura do Android dos tópicos anteriores. Esse layout define a imagem que será animada, e dois botões para criar a animação utilizando um arquivo em XML ou a API.

Para começar a brincadeira, veja o seguinte exemplo genérico de animator:

 `/res/anim/animator_1_para_0.xml`

```
<animator xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="1000"
    android:valueFrom="1" android:valueTo="0"
    android:valueType="floatType" android:repeatCount="1" android:repeatMode="reverse" />
```

Verificamos que essa animação define que o valor inicial é 1 e o final é 0, mas não define qual a propriedade ou tipo da animação que será realizado. Isso acontece porque o conceito de animações nesse novo framework é genérico. Podemos informar o valor inicial e final de forma abstrata, para posteriormente vincular esses valores com alguma propriedade real.

Uma característica importante que podemos verificar nesse recurso de animação é que o tempo da animação foi definido como um segundo, e, portanto, durante esse intervalo, o valor escolhido passará de 1 a 0 (note que os números são float).

Uma vez que o recurso de animação esteja criado, podemos recuperar essa animação e instanciar um objeto do tipo `ValueAnimator` da seguinte forma:

```
ValueAnimator a = (ValueAnimator) AnimatorInflater.loadAnimator(this,
    R.anim.animator_1_para_0);
```

Essa configuração de animação informa o tempo e os valores, mas não define quais as propriedades que precisam ser alteradas. Portanto, para aplicar a animação devemos criar um listener para ficar ouvindo o seu andamento, que durante determinado tempo vai nos enviar números de 1 até 0. Tal listener pode receber os valores e aplicá-los em algum objeto durante aquele intervalo de tempo, conforme demonstrado a seguir:

```
final ImageView img = ...
ValueAnimator a = (ValueAnimator) AnimatorInflater.loadAnimator(this,
    R.anim.animator_1_para_0);
a.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
    public void onAnimationUpdate(ValueAnimator animation) {
        // Fica ouvindo os valores durante a animação
        Float valor = (Float) animation.getAnimatedValue();
        // Altera o alpha
        img.setAlpha(valor);
    }
});
```

Note que no código o método `onAnimationUpdate(animation)` será chamado várias vezes durante o tempo que a animação estiver executando. Chamando o método `getAnimatedValue()`, recuperamos os valores que vão variar de 1.0 a 0.0. Feito isso, podemos fazer qualquer coisa, neste caso, estamos manualmente alterando a propriedade `alpha` da view, o que vai criar uma animação do tipo `fade_out`, ou seja, a view vai ficar transparente até desaparecer.

Como pudemos verificar, o novo framework de animação consiste em definir um conjunto de valores que serão aplicados durante determinado intervalo de tempo, mas cabe ao desenvolvedor utilizar um listener para receber esses valores e fazer a atualização necessária em seu objeto.

A ideia é legal, mas na prática acaba sendo trabalhosa demais. Mas espere: esse é o conceito que está por de baixo dos panos do framework, mas no dia a dia vamos trabalhar com a classe `ObjectAnimator`, que vai facilitar nossa vida.

9.14 Classe ObjectAnimator

Conforme vimos no exemplo anterior, tivemos um pouco de trabalho, pois foi necessário alterar a propriedade `alpha` da view manualmente.

Para facilitar a vida do desenvolvedor, foi criada a classe `ObjectAnimator`, que é uma subclasse de `ValueAnimator`, portanto faz tudo que ela faz, mas altera automaticamente determinada propriedade. Com a classe `ObjectAnimator`, criar uma animação do tipo `fade_out`, variando o `alpha` de 1 para 0, é simples assim:

```
ImageView img = ...
ObjectAnimator a = ObjectAnimator.ofFloat(img, "alpha", 1f, 0f);
anim.setDuration(2000);
a.start();
```

O método está criando uma configuração de animação que vai iniciar em 1 e terminar em 0, durante dois segundos. Desta vez, porém, não precisamos criar nenhum listener, pois a classe `ObjectAnimator` recebe como parâmetro a propriedade que deve ser alterada durante a animação, que nesse caso é a `alpha`. Internamente essa classe faz o mesmo processo manual que fizemos no exemplo anterior.

Existem várias propriedades que podemos alterar, outro exemplo seria mover a view 100 pixels para a direita, conforme demonstrado a seguir:

```
ImageView img = ...
ObjectAnimator a = ObjectAnimator.ofFloat(img, "x", img.getX(), 100);
anim.setDuration(1000);
a.start();
```

Note que anteriormente tínhamos as classes `AlphaAnimation` e `TranslateAnimation`, para implementar as mesmas coisas. Mas agora uma única classe chamada `ObjectAnimator` é capaz de alterar qualquer propriedade do objeto, e ainda manter o estado do objeto, o que era um bug no framework antigo.

Para a classe `ObjectAnimator` conseguir alterar as propriedades do objeto, temos uma restrição: o objeto que receberá a animação precisa ter os métodos `get` e `set` das propriedades desejadas. Por exemplo, ao criar uma animação para a propriedade `alpha`, podemos utilizar o seguinte código:

```
ObjectAnimator a = ObjectAnimator.ofFloat(img, "alpha", 1f, 0f);
```

Significa que os métodos `getAlpha()` e `setAlpha(float)` existem na classe `View`. Note que o método `ofFloat` indica que o tipo do parâmetro é `float`. Seguindo o mesmo princípio, sabemos que podemos alterar a propriedade `x`, porque existem os métodos

`getX()` e `setX(float)`. É dessa maneira que a classe `ObjectAnimator` funciona, podendo alterar qualquer propriedade de um objeto durante a animação em determinado intervalo de tempo. É um conceito bem flexível.

Nota: para criar as animações, estamos usando o método `ObjectAnimator.ofFloat`, porque os parâmetros que estamos tentando manipular são do tipo `float`, mas também existem os métodos `ObjectAnimator.ofInt` e `ObjectAnimator.ofObject`, que permitem manipular outros tipos.

A lista a seguir exhibe as propriedades mais comuns que podemos alterar com essa classe, para obter as tradicionais animações de `alpha`, `rotate`, `scale`, `translate`.

`alpha`

Transparência da `view`, que pode variar entre 0 e 1.

`x` e `y`

Coordenadas com a posição da `view`.

`translationX` e `translationY`

Essas propriedades representam um deslocamento segundo as coordenadas `x` e `y`. Por exemplo, se uma `view` tiver a propriedade `translationX=-50`, ela será deslocada 50 pixels para a esquerda. Isso é útil para colocar layouts fora da tela, e depois animá-los para dentro.

`rotation`, `rotationX` e `rotationY`

Propriedades sobre a rotação da `view` em graus.

`scaleX` e `scaleY`

Propriedades para controlar a escala para redimensionar uma `view`.

`pivotX` e `pivotY`

Propriedades que definem o eixo utilizado nas animações de rotação e escala, em que o padrão é o centro da `view`.

Agora que conhecemos a classe `ObjectAnimator`, vamos refazer a maioria dos exemplos que estudamos até o momento, para você verificar a simplicidade desse framework de animação.

Nota: o framework Property Animations funciona somente no Android 3.0 (Honeycomb) ou superior. Se você deseja utilizar as facilidades desse framework em versões antigas do Android, procure pela biblioteca NineOldAndroids (<http://nineoldandroids.com/>). Eu particularmente utilizo essa biblioteca em meus aplicativos.

9.15 ObjectAnimator – animação fade_in/fade_out

Neste próximo exemplo, vamos implementar os famosos efeitos de fade_in e fade_out, com o objetivo de alterar a transparência da view.



AlphaAnim.java

```
public class AlphaAnim extends Activity {
    private boolean visivel = true;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.exemplo_animacao);
    }

    public void onClickAnimarXML(View view) {
        ImageView img = (ImageView) findViewById(R.id.img);
        ObjectAnimator a = (ObjectAnimator) AnimatorInflater.loadAnimator(this, R.anim.fade_out);
        a.setTarget(img);
        animar(a);
    }

    public void onClickAnimarAPI(View view) {
        ImageView img = (ImageView) findViewById(R.id.img);
        ObjectAnimator a = ObjectAnimator.ofFloat(img, "alpha", 1f, 0f);
        animar(a);
    }

    private void animar(ObjectAnimator anim) {
        anim.setDuration(2000);
        if(visivel) {
            anim.start();
        } else {
            // Apenas reverte a animação
            anim.reverse();
        }
        // Inverte o flag para na próxima vez utilizar a animação inversa
        visivel = !visivel;
    }
}
```

Esse exemplo utiliza a classe `ObjectAnimator` para criar a animação pela API e por XML. Para criar o mesmo efeito em XML, basta criar um arquivo que comece com a tag `<objectAnimator>` e configurar as propriedades.

 /res/anim/fade_out.xml

```
<objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"
    android:propertyName="alpha"
    android:valueFrom="1"    android:valueTo="0"
    android:duration="1000" android:valueType="floatType" />
```

Ao executar esse exemplo, o resultado será a animação de `fade_in` e `fade_out`, como o esperado. A classe `ObjectAnimator` facilita bastante a criação da animação. Uma das funcionalidades mais interessantes é o método `reverse()` disponível na classe `Animator`, que faz com que uma animação execute no sentido inverso, desfazendo-se. Esse método é um espetáculo porque não precisamos criar duas animações distintas, uma para fazer o objeto aparecer e outra para desaparecer. Basta chamar o método `reverse()` e a animação é executada no sentido contrário. Compare essa implementação com a anterior e veja a diferença.

Nota: o método `reverse()` reverte a animação tornando o código bem mais simples. Nos casos em que você precise programar várias animações em conjunto (ex: `alpha` + `translate` + `rotate`) o método `reverse()` realmente vale a pena, pois economiza várias linhas de código.

9.16 ObjectAnimator – animação de movimento

Para mostrar como o framework Property Animation é simples, vamos refazer os outros exemplos de animações que já demonstramos neste capítulo.

O seguinte código mostra como mover a view para baixo, algo que no framework View Animation seria feito com a classe `TranslateAnimation`.

```
ImageView img = (ImageView) findViewById(R.id.img);
// Pela API
ObjectAnimator a = ObjectAnimator.ofFloat(img, "y", 50, 200); // De y 50 até 100
// Por XML
ObjectAnimator a = (ObjectAnimator) AnimatorInflater.loadAnimator(this, R.anim.mover_baixo);
```

Neste exemplo, estamos movendo a view para baixo, alterando a propriedade `y`.

 /res/anim/mover_baixo.xml

```
<objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"
    android:propertyName="y"
    android:valueFrom="50" android:valueTo="200"
    android:duration="1000" android:valueType="floatType" />
```

Nota: execute o projeto PropertyAnimation disponível nos exemplos deste capítulo para conferir o resultado das animações.

9.17 ObjectAnimator – animação de rotação

O código a seguir mostra como girar a view em um ângulo de 360°, algo que no framework **View Animation** seria feito com a classe `RotateAnimation`.

```
ImageView img = (ImageView) findViewById(R.id.img);
// Pela API
ObjectAnimator a = ObjectAnimator.ofFloat(img, "rotation", 0, 360);
// Por XML
ObjectAnimator a = (ObjectAnimator) AnimatorInflater.loadAnimator(this, R.anim.rotate_360);
```

No caso do arquivo XML, ele pode ser criado desta forma:

 /res/anim/rotate_360.xml

```
<objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"
    android:propertyName="rotation"
    android:valueFrom="0" android:valueTo="360"
    android:duration="1000" android:valueType="floatType" />
```

Nota: no framework `View Animation`, vimos que o código necessário para animar uma view muitas vezes era extenso e justamente por isso era recomendado criar a animação em XML para facilitar a leitura do código. Com o framework `Property Animation` o código se tornou extremamente amigável, e podemos criar animações com apenas uma linha de código. Portanto, fica a seu critério utilizar API ou XML.

9.18 ObjectAnimator – animação de escala

Até o momento, trabalhamos com a classe `ObjectAnimator` para alterar uma única propriedade de um objeto para criar a animação. No entanto, às vezes, dependendo do caso, precisamos alterar mais de uma propriedade.

Neste próximo exemplo precisamos alterar duas propriedades do objeto, `scaleX` e `scaleY`, pois precisamos criar o efeito de diminuir o objeto até ele desaparecer, e depois aumentá-lo novamente até o tamanho normal. Para alterar mais de uma propriedade, é necessário criar um `PropertyValuesHolder` com as informações de que precisamos. Essa classe apenas tem o objetivo de armazenar a informação sobre uma propriedade para ser animada.

```
PropertyValuesHolder anim1 = PropertyValuesHolder.ofFloat("scaleX", 1, 0);
PropertyValuesHolder anim2 = PropertyValuesHolder.ofFloat("scaleY", 1, 0);
```

Depois de criar essas duas configurações, podemos agrupá-las e criar um `ObjectAnimator` utilizando o método `ofPropertyValuesHolder(objeto, valores...)`, no qual podemos informar várias instâncias de `PropertyValuesHolder`.

```
ObjectAnimator a = ObjectAnimator.ofPropertyValuesHolder(img, anim1, anim2);
```

Feito isso, o objeto `ObjectAnimator` vai executar as duas animações simultâneas para criar o efeito de escala tanto no eixo x quanto y do objeto.

9.19 AnimatorSet – criando um conjunto de animações

A classe `AnimatorSet` pode ser útil para criar um conjunto de animações que podem ser executadas em sequência ou ao mesmo tempo. A seguir, podemos visualizar um trecho de código que mostra como utilizar a classe `AnimatorSet`:

```
ImageView img = (ImageView) findViewById(R.id.img);
// Cria as animações
ObjectAnimator alphaAnim = ObjectAnimator.ofFloat(img, "alpha", 1f, 0f);
ObjectAnimator translateAnim = ObjectAnimator.ofFloat(img, "y", y, img.getHeight()*2);
// Insere na lista
AnimatorSet lista = new AnimatorSet();
lista.playTogether(translateAnim, alphaAnim);
// Dispara a animação
lista.setDuration(2000);
lista.start();
```

Caso você prefira utilizar o XML, basta inflar a lista de animações desta forma:

```
ImageView img = (ImageView) findViewById(R.id.img);
AnimatorSet lista = (AnimatorSet) AnimatorInflater.loadAnimator(this, R.anim.animator_set);
```

```

 /res/anim/animator_set.xml

<?xml version="1.0" encoding="utf-8"?>
<set>
  <objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="1000"
    android:valueTo="200" android:valueType="floatType"
    android:propertyName="x" android:repeatCount="1" />
  <objectAnimator xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="1000"
    android:valueTo="400" android:valueType="floatType"
    android:propertyName="y" android:repeatCount="1" />
</set>

```

Nota: recomendo executar o projeto de exemplo deste capítulo para você visualizar as animações. Os exemplos do projeto são coerentes com os títulos dos tópicos do livro.

9.20 AnimatorListener

Com o framework Property Animation também podemos monitorar a execução das animações, basta implementar a interface `android.animation.Animator.AnimatorListener`:

```

Animator anim = ...;
anim.addListener(new Animator.AnimatorListener() {
    @Override
    public void onAnimationStart(Animator animation) { }
    @Override
    public void onAnimationEnd(Animator animation) { }
    @Override
    public void onAnimationCancel(Animator animation) { }
    @Override
    public void onAnimationRepeat(Animator animation) { }
});

```

Acredito que os métodos dessa interface sejam autoexplicativos. Uma vantagem dessa API é que, se você quiser, podemos usar a classe `AnimatorListenerAdapter` que já implementa a interface `AnimatorListener` e deixa os métodos vazios, de modo que precisaremos apenas sobrescrever o método desejado.

```

Animator anim = ...;
anim.addListener(new AnimatorListenerAdapter() {

```

```

@Override
public void onAnimationEnd(Animator animation) {
    // fim da animação
}
});

```

Nota: a interface `AnimatorListener` do novo framework de animação do Android 3.x é muito parecida com a interface `AnimationListener` que estudamos anteriormente para o Android 2.x. Nessa nova interface, porém, os métodos recebem como parâmetro um objeto do tipo `Animator` e não um `Animation`.

9.21 ViewPropertyAnimator – animação do jeito fácil

Para o final do capítulo deixei a cereja do bolo, pois a classe `ViewPropertyAnimator` permite criar animações com uma linguagem simples e alterando várias propriedades ao mesmo tempo.

Se quisermos mover as propriedades `x` e `y` de uma `view` ao mesmo tempo, poderíamos utilizar um código assim:

```

ObjectAnimator animX = ObjectAnimator.ofFloat(myView, "x", 50f);
ObjectAnimator animY = ObjectAnimator.ofFloat(myView, "y", 100f);
AnimatorSet animSetXY = new AnimatorSet();
animSetXY.playTogether(animX, animY);
animSetXY.start();

```

Com a nova sintaxe, basta uma linha de código e a animação está feita:

```
myView.animate().x(50f).y(100f).alpha(0);
```

O método `animate()` da classe `View` retorna um objeto do tipo `ViewPropertyAnimator` que contém vários métodos de conveniência para configurar as propriedades da animação. Esse tipo de sintaxe de programação é chamada de interface *fluent* (fluent interface), pois em uma única linha de código e com uma linguagem natural é possível configurar várias propriedades do objeto de forma aninhada.

Dica: a classe `ViewPropertyAnimator` somente está disponível no Android 3.1 (API Level 12), mas caso você precise utilizar o framework `Property Animation` em versões anteriores do Android, procure pela biblioteca `NineOldAndroids` (<http://nineoldandroids.com/>).

9.22 Classe ValueAnimator – outro exemplo

Como já expliquei anteriormente, a classe `ValueAnimator` é a base do novo framework, mas com ela temos o trabalho manual de obter os dados e atualizar as propriedades na view. A classe `ObjectAnimator` foi criada para facilitar esse trabalho. Se precisar, releia as explicações anteriores sobre essas classes.

Às vezes, no entanto, o que você quer é justamente obter todos os valores numéricos durante um intervalo de tempo. Para exemplificar, o Google Fit utiliza o pedômetro (sensor de passos) e mostra uma animação assim que a tela é aberta. Nessa animação a quantidade de passos é atualizada de forma sequencial e animada. Então se você deu 1.000 passos e fechou o aplicativo, e depois deu mais 500 passos e abriu o aplicativo, o Google Fit vai mostrar uma animação para atualizar o texto do `TextView` de 1.000 até 1.500.

Como podemos fazer esse tipo de animação? A classe `ValueAnimator` pode ajudar, basta definir o intervalo numérico que você deseja e que o framework envie as atualizações e o tempo da animação. O seguinte trecho de código mostra como atualizar o texto de um `TextView` do valor 0 para 100 de forma incremental e animada, durante o intervalo de um segundo. Você poderá ver os números incrementando até chegar no valor final.

```
final TextView textView = ?
// Animação genérica de 1 até 100
ValueAnimator a = ValueAnimator.ofFloat(1, 100);
a.addUpdateListener(new ValueAnimator.AnimatorUpdateListener() {
    public void onAnimationUpdate(ValueAnimator animation) {
        // Recebe o valor e atualiza o texto.
        Float valor = (Float) animation.getAnimatedValue();
        textView.setText(String.valueOf(valor.intValue()));
    }
});
a.setDuration(1000);
a.setTarget(textView);
a.start();
```

9.23 Aplicando animações no layout

Outro recurso interessante é aplicar uma animação que foi definida na pasta `/res/anim` no layout da tela. A animação precisa ser do tipo `LayoutAnimationController` e pode ser definida em XML na pasta `/res/anim`.

A classe `LayoutAnimationController` serve para animar um gerenciador de layout `ViewGroup` durante a sua criação, de forma que a animação seja aplicada sucessivamente em todas as suas views filhas, durante a criação do layout. O exemplo a seguir mostra como animar o layout da tela. Note que o arquivo começa com a tag `<layoutAnimation>` e utiliza uma animação já existente para ser aplicada. Escolhemos a animação `/res/anim/fade_in.xml` para que cada view apareça na tela aos poucos, uma após a outra.

 `/res/anim/animacao_layout_fade_in.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<layoutAnimation xmlns:android="http://schemas.android.com/apk/res/android"
    android:animation="@anim/fade_in"
    android:duration="50" />
```

Depois de criar a animação de layout, basta utilizá-la em qualquer layout criado na pasta `/res/layout`, conforme demonstrado a seguir:

```
<LinearLayout ... android:layoutAnimation="@anim/animacao_layout_fade_in" >
```

Outro recurso interessante de animação suportado pelo Android são as animações que executam quando o layout de algum gerenciador de container é alterado, como, por exemplo, uma view ser adicionada ou removida dinamicamente.

Por padrão, o Android não faz esse tipo de animação, mas para habilitá-la basta configurar o atributo `android:animateLayoutChanges` para `true`, conforme demonstrado a seguir:

```
<LinearLayout android:animateLayoutChanges="true" ... />
```

Pronto, agora como um passe de mágica sempre que você adicionar ou remover uma view desse layout o Android vai animar essa transição.

Nota: para conferir os exemplos de animação de layout, execute o projeto `LayoutAnimations` disponível nos exemplos deste capítulo no Android Studio.

9.24 Aplicando animações nos fragments

Depois de estudar as animações, fica fácil aplicá-las em qualquer lugar do sistema.

Como fragments são muito utilizados para adicionar, substituir e remover componentes do layout, podemos aplicar uma animação ao executar a `FragmentManager`. O código-fonte a seguir mostra como substituir um fragment no layout de forma animada, utilizando as animações de `fade_in` e `fade_out` que controlam a transparência das views.

```

android.support.v4.app.FragmentManager fm = getSupportFragmentManager();
FragmentManager ft = fm.beginTransaction();
ft.setCustomAnimations(android.R.anim.fade_in, android.R.anim.fade_out);
Fragment1 frag1 = new Fragment1();
ft.replace(R.id.layoutFrag, frag1, "Fragment1");
ft.commit();

```

O método `setCustomAnimations(int animEnter, int animExit)` é utilizado para especificar a animação de entrada do novo fragment, e a animação de saída do fragment atual, em caso de substituição. Neste exemplo utilizei a classe `android.R` nativa, mas se você quiser é possível criar suas próprias animações.

Para testar esse código, abra o projeto **Fragments-ActionBarTabs** no Android Studio. Esse é o mesmo exemplo que fizemos com fragments + tabs no capítulo 8, sobre fragments. Porém, ao clicar em uma tab, estou substituindo o fragment utilizando uma animação, confira!

9.25 Aplicando animações ao navegar entre activities

A partir do Android 4.1 (Jelly Bean), podemos customizar a animação de navegação entre as activities. Para isso foi criada a classe `ActivityOptions` com três métodos:

`makeCustomAnimation(Context context, int enterResId, int exitResId)`

Configura uma animação customizada para a activity que está entrando e saindo.

`makeScaleUpAnimation(View source, int startX, int startY, int width, int height)`

Configura uma animação customizada de escala para a activity que vai ser chamada. As coordenadas `startX` e `startY` são as posições para iniciar a animação, referentes à view. Os parâmetros `width` e `height` definem o tamanho inicial da nova activity.

`makeThumbnailScaleUpAnimation(View source, Bitmap thumbnail, int startX, int startY)`

Configura uma animação customizada de escala para a activity, mas o objeto que dá origem à animação é uma imagem de `Bitmap`, e as coordenadas `startX` e `startY` são as posições para iniciar a animação, referentes ao `bitmap`.

Para brincarmos com as animações de transição entre telas, crie um novo projeto chamado **HelloActivityTransition**. No layout da activity, vamos adicionar a imagem do planeta Terra posicionada no centro. Deixe o tamanho da figura com 100dp.

```

<?xml /> /res/layout/activity_main.xml

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <ImageView android:id="@+id/img" android:layout_gravity="center"
        android:src="@drawable/planeta_03_terra"
        android:layout_width="100dp" android:layout_height="100dp"
        android:onClick="onClickPlaneta" />
</FrameLayout>

```

A ideia é que, ao clicar na figura do planeta Terra, a activity `PlanetaActivity` seja chamada com uma animação de transição. A segunda activity contém o mesmo layout, porém a foto do planeta está maior. Para fazer a transição, em vez de chamar o método `startActivity(intent)` como de costume, podemos utilizar a classe `ActivityOptionsCompat` e o método `makeCustomAnimation(context, animacaoEntradaId, animacaoSaidaId)` conforme demonstrado a seguir.

```

<?xml /> MainActivity.java

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void onClickPlaneta(View view) {
        Intent intent = new Intent(getBaseContext(), PlanetaActivity.class);
        ActivityOptionsCompat opts =
            ActivityOptionsCompat.makeCustomAnimation(this, R.anim.fade_in, R.anim.fade_out);
        ActivityCompat.startActivity(this, intent, opts.toBundle());
    }
}

```

No código foi definida a animação `fade_in` (aparecer) para a activity que vai abrir e `fade_out` (desaparecer) para a activity que vai fechar, portanto crie os seguintes arquivos de animação na pasta `/res/anim`.

```

<?xml /> /res/anim/fade_in.xml

<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
    android:fromAlpha="0.0" android:toAlpha="1.0" android:duration="1000" />

```

 /res/anim/fade_out.xml

```
<?xml version="1.0" encoding="utf-8"?>
<alpha xmlns:android="http://schemas.android.com/apk/res/android"
    android:fromAlpha="1.0" android:toAlpha="0.0" android:duration="1000" />
```

Na segunda activity, vamos mostrar a mesma figura do planeta, sendo assim, você pode até copiar o layout anterior. A diferença é que na primeira activity deixei a figura pequena com 100dp e na segunda deixei a figura grande ocupando o tamanho que ela tem com a notação `wrap_content`. Na segunda activity, o detalhe importante é que sobrescrevemos o método `finish()`, pois é preciso customizar a animação de saída chamando o método `overridePendingTransition(animEntrada, animSaida)`.

 PlanetaActivity.java

```
public class PlanetaActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_planeta);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    }
    @Override
    public void finish() {
        super.finish();
        // Customiza a animação ao fechar a activity
        overridePendingTransition(R.anim.fade_in, R.anim.fade_out);
    }
}
```

A figura 9.7 mostra o resultado ao executar esse exemplo. Ao clicar na foto do planeta Terra, a figura vai sumir e aparecer devido às animações de transparência: `fade_in` e `fade_out`. Esse efeito é muito interessante e amigável aos olhos do usuário.

Se você quiser, é possível criar outros tipos de animações. É bem comum encontrar aplicativos que utilizam o efeito que desliza a tela da esquerda para a direita. Isso pode ser feito com os seguintes arquivos de animação.

 /res/anim/slide_in_left.xml

```
<translate xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="500" android:fromXDelta="100%p" android:toXDelta="0" />
```

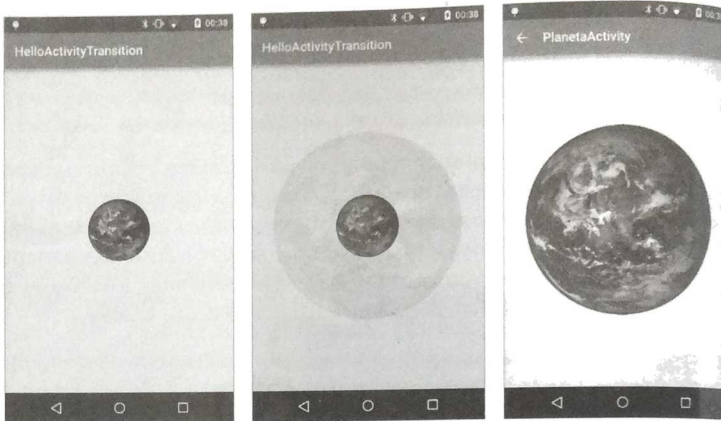


Figura 9.7 – Animação de transição com transparência.

 /res/anim/slide_out_left.xml

```
<translate xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="500" android:fromXDelta="0" android:toXDelta="-50%p" />
```

No código da MainActivity, basta utilizar esses arquivos de animação para fazer a transição de telas com um efeito deslizante.

```
public void onClickPlaneta(View view) {
    Intent intent = new Intent(getApplicationContext(), PlanetaActivity.class);
    ActivityOptionsCompat opts =
        ActivityOptionsCompat.makeCustomAnimation(this, R.anim.slide_in_left,
            R.anim.slide_out_left);
    ActivityCompat.startActivity(this, intent, opts.toBundle());
}
```

Só tenha atenção, pois se estamos fazendo a animação entrar com o efeito de deslizar da direita para a esquerda, temos de fazer o contrário ao pressionar o botão voltar, ou seja, temos de sair da tela deslizando da esquerda para a direita. portanto crie os seguintes arquivos de animações:

 /res/anim/slide_in_right.xml

```
<translate xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="500" android:fromXDelta="-50%p" android:toXDelta="0" />
```

 /res/anim/slide_out_right.xml

```
<translate xmlns:android="http://schemas.android.com/apk/res/android"
    android:duration="500" android:fromXDelta="0" android:toXDelta="100%p" />
```

Feito isso, utilize estas animações na classe `PlanetaActivity`:

```
@Override
public void finish() {
    super.finish();
    // Para voltar utiliza a animação da esquerda para a direita
    overridePendingTransition(R.anim.slide_in_right, R.anim.slide_out_right);
}
```

A figura 9.8 mostra o resultado da animação. Desta vez, ao clicar na imagem do planeta Terra, a segunda activity entrou na tela da direita para a esquerda. Ao clicar no botão voltar, a animação é o inverso.

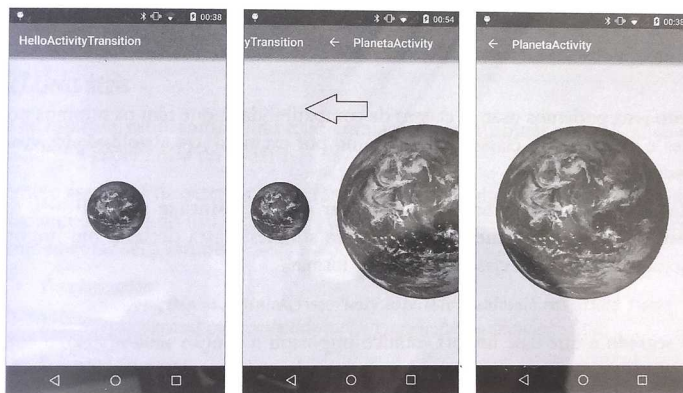


Figura 9.8 – Animação de transição com deslize.

Caso queira conferir outras animações, criadas com os métodos `makeScaleUpAnimation(...)` e `makeThumbnailScaleUpAnimation(...)` abra o projeto **ActivityAnimation-GoogleDemo** disponível nos exemplos do livro.

Nota: a classe `ActivityOptions` foi criada no Android 4.1, mas para facilitar nossa vida o Google disponibilizou as classes `android.support.v4.app.ActivityCompat` e `android.support.v4.app.ActivityOptionsCompat` para termos a compatibilidade com versões antigas. Veja que estamos utilizando as classes de compatibilidade no código. Caso a animação não possa ser feita nas versões antigas do Android, a activity vai abrir normalmente.

9.26 NineOldAndroids – animações com compatibilidade

Lá vai a dica final sobre animações! Neste capítulo, estudamos o framework **Property Animations** que funciona somente no Android 3.0 (Honeycomb) ou superior e facilita bastante a criação de animações. E o melhor exemplo de todos foi o que mostramos no tópico **ViewPropertyAnimator**, pois com uma única linha de código é possível animar diversas propriedades de uma view.

```
myView.animate().x(50f).y(100f).alpha(0);
```

Mas como usamos essa sintaxe simplificada em versões antigas do Android?

A resposta é a biblioteca **NineOldAndroids** (<http://nineoldandroids.com/>). Para utilizá-la, adicione esta dependência no arquivo *app/build.gradle*.

```
 /app/build.gradle

dependencies {
    ...
    compile 'com.nineoldandroids:library:2.4.0'
}
```

Feito isso, podemos usar as classes de compatibilidade que têm os mesmos nomes e métodos das classes nativas, como por exemplo: `com.nineoldandroids.view.ViewPropertyAnimator`.

Mas o que eu realmente gosto de fazer é usar a sintaxe simplificada da **ViewPropertyAnimator**. Felizmente, isso é muito simples; basta declarar um import estático no código da classe da seguinte forma:

```
import static com.nineoldandroids.view.ViewPropertyAnimator.animate;
```

O segredo é que esse import estático importou a função `animate(view)`, e com ela podemos animar qualquer view utilizando a sintaxe resumida. O seguinte exemplo mostra como mover uma imagem no eixo x e y, ao mesmo tempo em que é rotacionada em 180°; no final, ficará com 50% de transparência, tudo em uma única linha de código.

```
animate(img).xBy(200).yBy(200).rotation(180).alpha(0.5f).setDuration(2000);
```

A figura 99 mostra o resultado do projeto de exemplo **NineOldAndroids** disponível nos exemplos do livro, o qual executa exatamente esse código mostrado anteriormente.

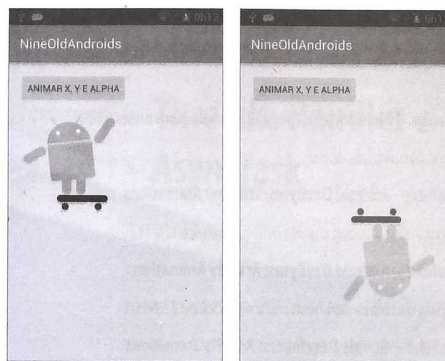


Figura 9.9 – Animação.

9.27 Links úteis

Neste capítulo, estudamos como criar animações para turbinar seu aplicativo e melhorar a experiência do usuário.

Como o assunto é de extrema importância para criar aplicativos fantásticos que encantem os olhos do usuário, separei vários links da documentação oficial para complementar seus estudos,

- **View Animation**

<http://developer.android.com/guide/topics/graphics/view-animation.html>

- **Property Animation**

<http://developer.android.com/guide/topics/graphics/prop-animation.html>

- **Animating Layout Changes**

<http://developer.android.com/training/animation/layout.html>

- **Drawable Animation**

<http://developer.android.com/guide/topics/graphics/drawable-animation.html>

- **Android Training – Animations**

<https://developer.android.com/training/animation/index.html>

- **Android Developers Blog – Introducing ViewPropertyAnimator**

<http://android-developers.blogspot.com.br/2011/05/introducing-viewpropertyanimator.html>

- **NineOldAndroids – Biblioteca de compatibilidade para animações**

<http://nineoldandroids.com/>

- **Video no YouTube – Android DevBytes: Window Animations**

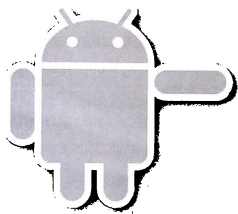
<https://www.youtube.com/watch?v=Ho8vk61lVIU>

- **Video no YouTube – Android DevBytes: Activity Animations**

<https://www.youtube.com/watch?v=CPxkoe2MraA>

- **Video no YouTube – Google Developers: Activity Transitions**

<https://www.youtube.com/watch?v=RhiPJByIMrM>



CAPÍTULO 10

Threads, Handler e AsyncTask

Neste capítulo, vamos estudar o conceito de threads e a classe `Handler`, a qual é utilizada para enviar ou agendar mensagens para serem executadas na thread principal da aplicação, conhecida como `Main Thread` ou `UI Thread`.

A classe `Handler` tem um papel importante na arquitetura do Android, porque somente por meio dela é possível atualizar a interface gráfica a partir de uma thread diferente da thread principal.

10.1 Introdução

Quando um aplicativo é aberto no Android, um processo dedicado no sistema operacional é criado para executá-lo. Cada processo tem uma única thread, conhecida como `Main Thread` ou `UI Thread`, responsável por gerenciar todos os eventos da tela, assim como atualizar a interface gráfica. O fato é que muitas vezes a `activity` pode realizar uma tarefa um pouco demorada, e para não travar a interface de usuário é recomendado que esse tipo de código seja executado em outra thread.

Podemos dizer que, sempre que uma consulta é realizada em um web service, banco de dados, agenda de contatos ou leitura de um arquivo, é obrigatório criar uma thread para desvincular esse processamento da thread principal. Antigamente isso era apenas uma recomendação, mas nas versões mais novas do Android, se o código fizer uma operação de I/O na thread principal, o sistema vai lançar a exceção `NetworkOnMainThreadException`.

Portanto, como regra, toda operação de I/O, seja consultar um web service, ler um arquivo ou acessar o banco de dados, deve executar em uma thread separada.

Outro motivo importante para utilizar threads é porque a thread principal da aplicação deve responder aos eventos do usuário, em no máximo em cinco segundos. Se esse tempo for ultrapassado, o erro ANR (Application Not Responding) será lançado. Esse erro é a clássica mensagem com um **Force Close** que aparece em muitas aplicações, porque, nesse caso, o Android entende que a aplicação não está respondendo e exibe esse alerta para o usuário fechá-la ou aguardar. Para evitar esse tipo de erro, é necessário utilizar threads.

Uma vez que já justificamos a necessidade de utilizar threads, vamos ver um trecho de código em Java que executa um código em uma nova thread.

```
new Thread() {  
    public void run() {  
        // Código que deve executar em segundo plano aqui  
    };  
}.start();
```

Uma thread deve ser filha da classe `Thread` e deve implementar o método `run()`. Ao chamar o método `start()`, a thread é iniciada, ou seja, o método `run()` vai executar em segundo plano. Para mais detalhes sobre threads no Java, recomendo uma leitura adicional em livros sobre essa linguagem.

No caso do Android, sempre que uma thread é iniciada, temos um problema, pois por questões de segurança e concorrência o Android não permite que uma thread diferente da principal atualize a interface gráfica da tela. Por isso, a classe `android.os.Handler` foi criada com o objetivo de enviar uma mensagem para a thread principal, para que, em algum momento apropriado, essa mensagem possa ser processada de forma segura e consequentemente atualizar a interface gráfica da tela (view). Um exemplo clássico de utilização de threads e atualização de interface gráfica com um `Handler` pode ser visto a seguir:

```
final Handler handler = new Handler;  
new Thread() {  
    public void run() {  
        // Código que deve executar em segundo plano aqui  
        handler.post(new Runnable() {  
            public void run() {  
                // Código que atualiza a interface aqui  
            }  
        });  
    };  
}.start();
```

Ou podemos utilizar o método `runOnUiThread(runnable)`, que é um atalho para utilizar um handler que está dentro da activity.

```
new Thread() {
    public void run() {
        // Código que deve executar em segundo plano aqui
        runOnUiThread(new Runnable() {
            public void run() {
                // Código que atualiza a interface/view aqui
            }
        });
    }
}.start();
```

Essa é a utilização clássica de um handler, que tem como objetivo sincronizar o código de uma thread para atualizar a interface. No entanto, o handler também é muito utilizado para agendar tarefas e enviar mensagens dentro da activity.

Concluindo, podemos dizer que existem dois bons motivos para usar a classe `android.os.Handler`:

1. Atualizar a interface (view) sempre que uma thread for utilizada para fazer algum processamento em segundo plano.
2. Agendar uma mensagem `android.os.Message` ou um `java.lang.Runnable` para executar em determinado momento. Essa mensagem pode ser enviada instantaneamente ou com um intervalo de tempo (delay). Cada mensagem enviada é processada em uma fila de mensagens única para cada handler, que está vinculada à thread principal da aplicação.

Resumo: no Android, cada aplicação é executada em um único processo, e cada processo por sua vez tem uma thread dedicada. Essa thread também é responsável por desenhar e tratar todos os eventos da interface gráfica, e é conhecida como Main Thread ou UI Thread. Existe uma regra no Android que diz que somente a UI Thread pode atualizar a interface, ou seja, somente ela pode chamar qualquer método que vai atualizar uma view. A classe `Handler` é utilizada para enviar uma mensagem para ser processada pela UI Thread, que geralmente é um código que vai atualizar a view.

10.2 Método `sendMessage(msg)`

Para enviar uma mensagem com a classe `Handler`, podemos utilizar o método `sendMessage(msg)` e suas variantes, conforme demonstrado a seguir:

Método	Descrição
<code>sendMessage(msg)</code>	Envia a mensagem informada para a fila de mensagens para ser processada assim que possível.
<code>sendEmptyMessage(int)</code>	Envia a mensagem contendo apenas o atributo <code>what</code> informado como parâmetro. É a mesma coisa que criar um objeto <code>android.os.Message</code> e configurar apenas o atributo <code>what</code> .
<code>sendMessageDelayed(msg, long)</code>	Envia a mensagem para a fila de mensagens, mas ela é processada somente depois de determinado tempo informado. O segundo argumento é do tipo <code>long</code> , que representa o tempo em milissegundos que a mensagem deve aguardar antes de ser enviada.
<code>sendMessageAtTime(msg, long)</code>	Envia a mensagem para a fila de mensagens, mas ela é processada somente na data informada. O segundo argumento é do tipo <code>long</code> , que representa uma data em milissegundos para executar a mensagem.

Para brincar com a classe `Handler`, vamos criar um exemplo para enviar uma mensagem com o método `sendMessageDelayed(msg, delay)`, o qual vai enviar a mensagem com atraso (`delay`) de três segundos. Para continuar, crie o projeto `HelloHandler`, ou se preferir abra o projeto de exemplo que acompanha o livro.

A seguir, podemos visualizar o código-fonte da `activity` e seu `layout`.

 `/res/layout/activity_demo_handler_message.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" android:padding="20dp" >
    <TextView
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="Disparar uma mensagem com atraso de 3 segundos" />
    <Button
        android:id="@+id/btEnviar"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Enviar mensagem" />
</LinearLayout>
```



DemoHandlerMessageActivity.java

```

public class DemoHandlerMessageActivity extends AppCompatActivity {
    protected static final int MENSAGEM_TESTE = 1;
    private Handler handler = new TesteHandler();
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_demo_handler_message);
        findViewById(R.id.btEnviar).setOnClickListener(new Button.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Cria a mensagem com delay de três segundos
                Message msg = new Message();
                msg.what = MENSAGEM_TESTE;
                // Envia a mensagem
                handler.sendMessageDelayed(msg, 3000);
            }
        });
    }
    // Handler utilizado para receber a mensagem (classe interna)
    private class TesteHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            // O atributo msg.what permite identificar a mensagem
            switch (msg.what) {
                case MENSAGEM_TESTE:
                    Toast.makeText(getApplicationContext(), "A mensagem chegou!", Toast.LENGTH_SHORT).show();
                    break;
            }
        }
    }
}

```

Note que, para utilizar o método `sendMessage(msg)` ou uma de suas variações, é preciso criar uma subclasse da classe `android.os.Handler`; justamente por isso, criamos a classe `TesteHandler`. Feito isso, para enviar uma mensagem ao handler, foi criado um objeto do tipo `android.os.Message` e no atributo `what` foi informado um valor, que contém uma constante para identificar a mensagem.

```

Message msg = new Message();
msg.what = MENSAGEM_TESTE;
handler.sendMessageDelayed(msg, 3000);

```

O método `sendMessageDelayed(msg, delayMillis)` recebe a mensagem e o tempo em milissegundos (`delay`) para atrasá-la. Nesse exemplo, depois de três segundos, o método `handleMessage(message)` da classe interna `TesteHandler` foi chamado. Nesse momento, o valor informado no atributo `what` é utilizado para identificar a mensagem (isso é útil caso exista mais de uma), conforme demonstrado a seguir:

```
public void handleMessage(Message msg) {  
    // O atributo msg.what permite identificar a mensagem  
    switch (msg.what) {  
        case MENSAGEM_TESTE:  
            Toast.makeText(ExemploHandler.this, "A mensagem chegou!", Toast.LENGTH_SHORT).show();  
            break;  
    }  
}
```

A figura 10.1 mostra o resultado desse código executando no emulador. Ao clicar no botão **Enviar Mensagem**, a mensagem é disparada, e depois de três segundos o alerta é exibido.

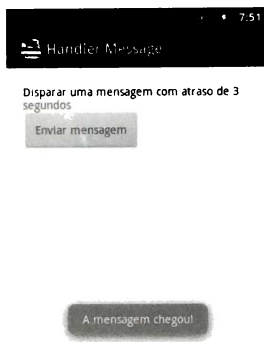


Figura 10.1 – Exemplo inicial da classe *Handler*.

Esse exemplo é extremamente simples, e no momento você pode não entender quais os reais benefícios de utilizar esse tipo de mensagem. Observe que nesse simples exemplo já podemos verificar que é possível agendar mensagens para execução posterior. Isso por si só já parece bastante útil. O fato é que nosso exemplo é muito simples para demonstrar o que a classe `android.os.Handler` pode fazer. Lembre-se de que as mensagens enviadas para um `Handler` são processadas pela `UI Thread`, e esse é o verdadeiro sentido de utilizar um handler. No momento,

vamos continuar com esses exemplos simples para você entender a sintaxe e como utilizar um `Handler` e mais para frente vamos analisar casos em que utilizar um `Handler` é obrigatório.

10.3 Método `post(runnable)`

Outra forma de enviar uma mensagem é com o método `postMessage(runnable)`, que funciona de forma similar ao método `sendMessage(msg)`, mas recebe uma implementação da interface `java.lang.Runnable`. A interface `Runnable` é bem conhecida pelos programadores Java e é utilizada para auxiliar na programação multithreading.

No caso do Android e a classe `Handler`, a interface `Runnable` também pode ser utilizada para enviar uma mensagem para a thread principal ou executar determinada tarefa com um tempo de atraso. A vantagem de utilizar um `java.lang.Runnable` em vez de enviar uma mensagem com a classe `android.os.Message` é que não é necessário criar uma subclasse de `android.os.Handler` e implementar o método `handleMessage(msg)`. Utilizando um `java.lang.Runnable`, naturalmente o método `run()` implementado por ela é chamado na thread principal.

Para executar ou agendar um `java.lang.Runnable`, são usados os mesmos métodos que para enviar uma mensagem, com os mesmos argumentos, mas agora os nomes dos métodos começam com a palavra `post(...)`. A seguir, veja a lista com a descrição de cada método:

Método	Descrição
<code>post(Runnable)</code>	Adiciona o <code>Runnable</code> na fila de mensagens.
<code>postDelayed(Runnable, long)</code>	Adiciona o <code>Runnable</code> na fila de mensagens, mas somente executa o processo depois do tempo especificado em milissegundos.
<code>postAtTime(Runnable, long)</code>	Adiciona o <code>Runnable</code> na fila de mensagens, mas somente executa o processo na data especificada em milissegundos.

Para praticar como utilizar o método `post(...)` e suas variações, vamos criar um exemplo para enviar uma mensagem com atraso (delay).



DemoHandlerMessageActivity.java

```
public class DemoHandlerRunnableActivity extends AppCompatActivity {  
    private Handler handler = new Handler();  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
    }  
}
```



```

setContentView(R.layout.activity_demo_handler_message);
findViewById(R.id.btEnviar).setOnClickListener(new Button.OnClickListener() {
    @Override
    public void onClick(View v) {
        // Cria a mensagem com delay de três segundos
        handler.postDelayed(new Runnable() {
            @Override
            public void run() {
                Toast.makeText(getBaseContext(), "A mensagem chegou com Runnable!",
                    Toast.LENGTH_SHORT).show();
            }
        }, 3000);
    }
});
}
}

```

Esse código tem o mesmo objetivo do exemplo anterior, mas note que a sintaxe utilizando um `Runnable` é mais simples do que enviar uma mensagem pelo `sendMessage(msg)`. Ao executar o código, o resultado será o mesmo da figura 10.1.

Vale lembrar que a interface `Runnable` é uma figurinha bem conhecida no mundo Java e podemos dizer que ela representa algum código que deve ser executado. Se necessário, procure mais detalhes sobre essa interface.

10.4 Atualizando a view dentro de uma thread

Depois dessa introdução sobre como utilizar a classe `Handler`, vamos entender quando ela é realmente necessária, que é quando as threads entram na brincadeira.

No capítulo 7 sobre a classe `View`, no exemplo sobre `ProgressDialog`, mostramos como fazer o download de uma imagem e mostrar uma mensagem de “por favor, aguarde” para o usuário. Se você voltar lá e ver o código desse exemplo, verá que ele utiliza threads e o método `runOnUiThread(runnable)` foi utilizado para enviar a mensagem à UI Thread. Antes de tudo, vale explicar que o método `runOnUiThread(runnable)` é um atalho ao método `post(runnable)` da classe `Handler`. Agora vamos estudar em mais detalhes porque isso foi necessário.

Vamos utilizar este código que vai funcionar como exemplo porque apresenta um bug.

```

res/layout/activity_download_imagem.xml

<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent" >
    <ImageView android:id="@+id/img" android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center" android:scaleType="fitCenter" />
    <ProgressBar android:id="@+id/progress"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="center"/>
</FrameLayout>

```

No layout XML existe um `ImageView` e por cima foi inserido um `ProgressBar` para mostrar a animação durante o download da imagem. O código da activity simplesmente faz o download de uma imagem e atualiza o conteúdo no `ImageView`.

DownloadImagemActivity.java

```

public class DownloadImagemActivity extends AppCompatActivity {
    private static final String URL = "http://livroandroid.com.br/imagens/livro01.png";
    private ProgressBar progress;
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_download_imagem);
        progress = (ProgressBar) findViewById(R.id.progress);
        progress.setVisibility(View.VISIBLE);
        downloadImagem(URL);
    }
    // Faz o download da imagem em uma nova thread
    private void downloadImagem(final String urlImg) {
        new Thread() {
            @Override
            public void run() {
                try {
                    // Faz o download da imagem
                    final Bitmap imagem = Download.downloadBitmap(urlImg);
                    // Atualiza a tela
                    atualizaImagem(imagem);
                } catch (IOException e) {
                    // Uma aplicação real deveria tratar este erro
                    Log.e("Erro ao fazer o download: ", e.getMessage(), e);
                }
            }
        }.start();
    }
}

```

```

    }
    }.start();
}
// Vai dar Erro neste método pois somente a UI Thread pode atualizar a view
private void atualizaImagem(final Bitmap imagem) {
    // Esconde o progress
    progress.setVisibility(View.INVISIBLE);
    // Atualiza a imagem
    ImageView imgView = (ImageView) findViewById(R.id.img);
    imgView.setImageBitmap(imagem);
}
}
}

```

O download é feito na classe `Download`, conforme demonstrado a seguir:

`Download.java`

```

public class Download {
    public static Bitmap downloadBitmap(String url) throws IOException {
        // Faz o download da imagem
        Bitmap bitmap = null;
        InputStream in = new URL(url).openStream();
        // Converte a InputStream do Java para Bitmap
        bitmap = BitmapFactory.decodeStream(in);
        in.close();
        return bitmap;
    }
}

```

Para o exemplo funcionar, declare a permissão `INTERNET` no arquivo *AndroidManifest.xml*.

`AndroidManifest.xml`

```

<manifest . . . />
    <uses-permission android:name="android.permission.INTERNET" />
    <application ... />
</manifest>

```

Para fazer o download, estamos iniciando corretamente uma thread, pois isso é obrigatório. Contudo, no final do download, ao tentar atualizar o `ImageView`, o Android vai lançar uma exceção, conforme mostra a figura 10.2.

Sempre que um erro acontece, você deve olhar os logs detalhados na janela `LogCat`, conforme a figura 10.3, que mostra a stack trace do erro. Podemos ver que o problema está na linha 46 da classe que criamos. Para mais detalhes do `LogCat`, revise

os capítulos 2 e 3. Note que a mensagem de erro é: “Only the original thread that created a view hierarchy can touch its views.” Traduzindo a mensagem, isso significa que somente a thread principal (UI Thread) pode atualizar as views! Portanto, é aqui que a classe `Handler` entra em ação.



Figura 10.2 – Erro ao executar o aplicativo.

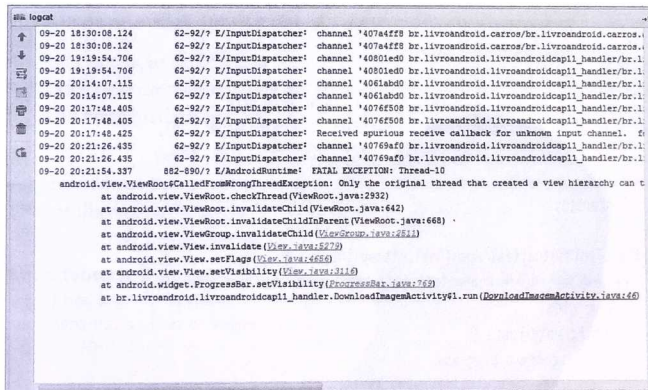


Figura 10.3 – Detalhes do erro no LogCat.

Como uma thread foi utilizada para fazer o download, é preciso utilizar um `Handler` para enviar a mensagem para a thread principal (UI Thread), a fim de atualizar a view. O código-fonte a seguir está atualizado para utilizar o método `runOnUiThread(runnable)`, que nada mais é do que um atalho para o método `Handler.post(runnable)` que estudamos anteriormente.

 DownloadImagemActivity.java

```

public class DownloadImagemActivity extends AppCompatActivity {
    private static final String URL = "http://livroandroid.com.br/imgs/livro_android.png";
    private ProgressBar progress;
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_download_imagem);
        progress = (ProgressBar) findViewById(R.id.progress);
        progress.setVisibility(View.VISIBLE);
        downloadImagem(URL);
    }
    // Faz o download da imagem em uma nova thread
    private void downloadImagem(final String urlImg) {
        new Thread() {
            @Override
            public void run() {
                try {
                    // Faz o download da imagem
                    final Bitmap bitmap = Download.downloadBitmap(URL);
                    // Atualiza a tela
                    atualizaImagem(imagem);
                } catch (IOException e) {
                    // Uma aplicação real deveria tratar este erro
                    Log.e("Erro ao fazer o download: ", e.getMessage(), e);
                }
            }
        }.start();
    }
    private void atualizaImagem(final Bitmap imagem) {
        runOnUiThread(new Runnable() { // Atualiza a view na UI Thread
            @Override
            public void run() {
                // Esconde o progress
                progress.setVisibility(View.INVISIBLE);
                // Atualiza a imagem
                ImageView ingView = (ImageView) findViewById(R.id.img);
                ingView.setImageBitmap(imagem);
            }
        });
    }
}

```

Com essa alteração, o exemplo vai executar perfeitamente, pois o download é feito em uma thread separada, mas a view é atualizada na thread principal com a ajuda do nosso amigo Handler. A figura 10.4 mostra o resultado.

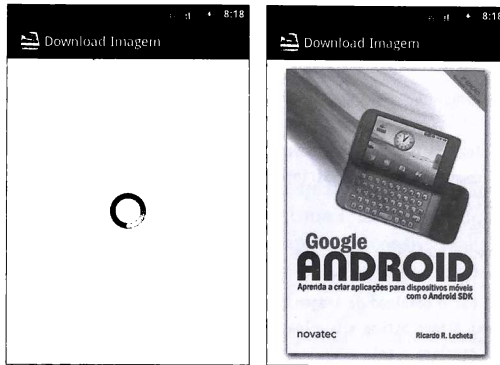


Figura 10.4 – Download da imagem realizado com sucesso.

Lembre-se de que estas duas tarefas são obrigatórias (é uma regra):

1. Toda operação de I/O, seja consultar um web service ou acessar o banco de dados, deve executar em uma thread separada. Qualquer processamento demorado também deve executar em sua própria thread para não travar a interface.
2. Somente a thread principal pode atualizar as views. Quando as threads secundárias terminam o seu trabalho, é necessário utilizar um handler ou o atalho `runOnUiThread(runnable)` para atualizar as views.

Importante: se a aplicação fechar por causa de um erro, olhe a exceção (stack trace) nos logs do LogCat. Mostrei o erro de propósito aqui apenas para você se acostumar a olhar os logs de erro.

10.5 Agendando tarefas contínuas na activity

A classe `Handler` é muito utilizada para executar tarefas de modo contínuo na activity. Por exemplo, se você precisar executar um código na activity a cada 30 segundos, isso pode ser feito com um handler. Isso é muito comum para criar telas que precisam ficar atualizando o seu conteúdo.

Para demonstrar, vamos alterar o exemplo de download da imagem para atualizar a imagem a cada dez segundos, ou seja, refazer o download continuamente. No código a seguir estou demonstrando apenas as partes importantes.



DownloadImagemActivity.java

```
public class DownloadImagemActivity extends AppCompatActivity {

    private Handler handler = new Handler();

    ...
    // Faz o download da imagem em uma nova thread
    private void downloadImagem(final String urlImg) {
        new Thread() {
            @Override
            public void run() {
                try {
                    // Faz o download da imagem...
                    final Bitmap bitmap = Download.downloadBitmap(URL);
                    // Atualiza a tela
                    atualizaImagem(imagem);
                    // Agenda o download novamente (daqui a dez segundos)
                    handler.postDelayed(new Runnable() {
                        public void run() {
                            downloadImagem();
                        }
                    }, 10000); // dez segundos
                } catch (IOException e) {
                    ...
                }
            }
        }.start();
    }

    ...
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // Cancela o runnable ao sair da activity
        handler.removeCallbacksAndMessages(null);
    }
}
```

Ao executar esse código, será feito o download da imagem normalmente. Depois de dez segundos, o download será feito novamente. Observe que no método `onDestroy()` da activity todas as mensagens enviadas ao handler foram canceladas

com o método `removeCallbacksAndMessages(null)`. Isso é necessário para garantir que nenhuma mensagem seja entregue com a activity fechada. Caso contrário, o handler continuaria executando continuamente sem a necessidade de fazer isso.

10.6 Implementação de um tela Splash Screen para sua aplicação

Provavelmente você já viu algum aplicativo que exibe uma tela inicial com a mensagem “Por favor, aguarde, carregando a aplicação...”, ou talvez exibindo uma imagem antes de carregar o aplicativo.

Essas telas de inicialização são chamadas de splash screen e para implementá-las no Android podemos utilizar um handler. Uma splash screen deve permanecer aberta por determinado tempo para que a aplicação consiga realizar algum processamento inicial. Enquanto isso, o usuário pode ficar observando uma imagem ou mensagem na tela.

Vamos criar uma splash screen na qual o layout da activity vai mostrar uma simples figura.

 /res/layout/activity_splash_screen.xml

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <ImageView android:src="@drawable/livro_android"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="center"/>
</FrameLayout>
```

O código da activity vai mostrar a tela por um segundo e depois vai prosseguir para a primeira activity do projeto. Para isso, uma mensagem é enviada ao handler com um atraso de um segundo. No momento em que o handler receber a mensagem, a aplicação já foi carregada e a próxima activity já pode ser exibida.

 SplashScreenActivity.java

```
public class SplashScreenActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Exibe o layout com a imagem...
        setContentView(R.layout.activity_splash_screen);
        // delay de 1 segundo
        new Handler().postDelayed(new Runnable() {
```



```

@Override
public void run() {
    // Inicia a MainActivity
    startActivity(new Intent(getBaseContext(), MainActivity.class));
    // Fecha a activity da splash
    finish();
}
},1000); // Um segundo de atraso
}
}
}

```

Uma splash screen geralmente é a primeira activity do projeto, que é aquela ação **MAIN** e categoria **LAUNCHER** configurada no **AndroidManifest.xml**. No projeto de exemplo deste capítulo, eu deixei a **SplashScreenActivity** como a primeira activity e cadastrei a **MainActivity** que mostra a lista com os exemplos como uma activity normal, conforme demonstrado a seguir.

AndroidManifest.xml

```

<manifest . . . >
    <uses-permission android:name="android.permission.INTERNET" />
    <application . . . >
        <activity android:name=".SplashScreenActivity" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:name=".MainActivity" android:label="@string/app_name" />
        . . .
    </application>
</manifest>

```

É por isso que, ao executar o projeto de exemplo deste capítulo, você verá uma splash screen antes de visualizar a lista com os exemplos. Lembre-se de que a tela de splash screen deve chamar o método **finish()** para encerrar essa activity para destruir a splash ao entrar no aplicativo.

Importante: saiba que, segundo as boas práticas de interface para Android, não é recomendado criar uma splash screen. Uma tela de splash screen deve ser utilizada somente se você obrigatoriamente precisa ganhar tempo antes de abrir a aplicação, por exemplo, para carregar informações necessárias para o seu aplicativo funcionar. O fato é que muitos aplicativos começaram a abusar da splash screen e mostram inclusive banners e anúncios antes de abrir o aplicativo.

Isso para os usuários é ruim, pois eles querem utilizar o aplicativo o mais rápido possível, e não ficar olhando figuras. Portanto, utilize uma splash screen somente se for realmente necessário. Esse assunto foi mostrado aqui apenas para você saber como implementar tal tipo de funcionalidade, mesmo porque está relacionado com o disparo de mensagens com atraso (delay) e a classe Handler.

10.7 AsyncTask

Criar uma thread é simples, mas vimos que é necessário utilizar um Handler ou o método de atalho `runOnUiThread(runnable)` para atualizar a interface.

Porém ao desenvolver aplicativos para Android não utilizamos threads diretamente, pois se recomenda utilizar a classe `AsyncTask`, a qual representa uma pequena biblioteca de threads. A seguir podemos visualizar as suas principais características.

1. A classe `AsyncTask` gerencia internamente as threads e handlers necessários para atualizar a interface.
2. Uma tarefa pode ser cancelada se chamar o método `cancell(boolean)`.
3. Contém métodos para atualizar o andamento (progresso) de uma tarefa, por exemplo, o progresso de um download.
4. Contém um pool de threads que pode executar as tarefas de modo serial ou em paralelo.

Para explicar o que é uma `AsyncTask`, vamos ver um exemplo de código. Basicamente para criar uma tarefa é preciso criar uma subclasse de `AsyncTask` e informar os três argumentos `<Params,Progress,Result>`, conforme demonstrado a seguir. A sintaxe `<Params,Progress,Result>` é das classes genéricas (Generics) do Java.

```
private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    @Override
    protected void onPreExecute() {
        // Executa na thread principal
        // Útil para mostrar um ProgressDialog ou ProgressBar
    }
    protected Long doInBackground(URL... urls) {
        // Executa em segundo plano (background)
        // O retorno do tipo "Long" é passado ao método onPostExecute()
        return 1L;
    }
    protected void onProgressUpdate(Integer... progress) {
        // Pode atualizar o progresso da tarefa
    }
}
```

```

        // O valor do progresso deve ser enviado via o método publishProgress(int)
        // durante o doInBackground()
    }
    protected void onPostExecute(Long result) {
        // Recebe o resultado do método doInBackground()
        // Executa na UI Thread e pode atualizar a view
    }
}

```

O código anterior está comentado, então leia-o com atenção. Depois de criar essa classe, para executá-la basta chamar o método `execute()` e informar os parâmetros, que nesse caso pela `Generics` foi definido como um objeto do tipo `URL`.

```
new DownloadFilesTask().execute(url);
```

Ou você pode até passar mais de um parâmetro:

```
new DownloadFilesTask().execute(url1, url2, url3);
```

Para vermos um exemplo mais detalhado, o código a seguir mostra como fazer o download de um arquivo, e ainda mostra como utilizar o método `onProgressUpdate(progress)`.

```

private class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            // Reporta o progresso
            publishProgress((int) ((i / (float) count) * 100));
            // Se o método cancel() foi chamado, termina.
            if (isCancelled())
                break;
        }
        return totalSize;
    }
    protected void onProgressUpdate(Integer... progress) {
        // Atualiza a interface com o progresso do download.
        setProgressPercent(progress[0]);
    }
    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}

```

Na prática, eu nunca precisei utilizar os métodos `publishProgress(progress)` e `onProgressUpdate(progress)`, pois geralmente em aplicativos a única coisa de que precisamos é consultar um web service, e durante isso podemos mostrar uma animação com um `ProgressBar` ou `ProgressDialog`. Portanto, costumo utilizar um template apenas com os métodos `onPreExecute()`, `doInBackground()` e `onPostExecute()`.

A lista a seguir explica o significado de cada método da `AsyncTask`.

Método	Descrição
<code>onPreExecute()</code>	Método executado antes de a thread iniciar, sendo uma boa oportunidade para exibir uma janela de progresso ao usuário ou uma mensagem de “por favor, aguarde”. Esse método executa na UI Thread.
<code>doInBackground()</code>	Método executado em background por uma thread, que deve conter todo o processamento pesado. Ele pode retornar um objeto qualquer, o qual será passado como parâmetro para o método <code>onPostExecute()</code> . É aqui que a thread executa, mas isso é feito automaticamente para você.
<code>onProgressUpdate()</code>	Método chamado na UI thread e recebe geralmente um inteiro para informar a quantidade do progresso. O progresso deve ser informado em background dentro do método <code>doInBackground()</code> . Para reportar o progresso, é utilizado o método <code>publishProgress(int)</code> .
<code>onPostExecute()</code>	Método executado na UI Thread, em que podemos atualizar a view com o resultado. Ele é chamado utilizando um Handler internamente.

Note que esses métodos não devem ser invocados manualmente, pois são chamados automaticamente pela classe `AsyncTask`. Portanto, para iniciar o processamento, é necessário apenas chamar o método `AsyncTask.execute(params...)`, informando os parâmetros se necessário, como por exemplo:

```
new DownloadFilesTask().execute(url1, url2, url3);
```

Nota: resumindo, você deve criar um `AsyncTask` e utilizar o método `doInBackground()` para fazer o processamento, o qual é executado automaticamente em uma thread. Quando terminar, utilize o método `onPostExecute()` para atualizar as views. A classe `AsyncTask` elimina a necessidade de criar uma thread e utilizar um handler, pois ela já faz esse trabalho para você.

Talvez a parte mais difícil de entender na classe `AsyncTask` sejam os seus três tipos genéricos, que têm a seguinte definição: `AsyncTask<Params, Progress, Result>`.

Parâmetro	Descrição
Params	O primeiro tipo genérico é chamado de Params, que são os argumentos que podemos passar ao método <code>execute(params...)</code> para executar o <code>AsyncTask</code> . No exemplo da classe <code>DownloadFilesTask</code> , o parâmetro foi definido como URL, e portanto a tarefa pode ser executada passando URLs como parâmetro ao método <code>execute()</code> , ex: <code>new DownloadFilesTask().execute(url1, url2, url3)</code> .
Progress	O segundo tipo genérico é chamado de Progress, e pode ser utilizado para receber um valor inteiro, que representa o progresso da execução, e em conjunto com uma barra de progresso, para notificar o usuário. No exemplo que criamos, esse parâmetro foi definido como <code>Integer</code> , e o método <code>onProgressUpdate(Integer... progress)</code> ficou com essa assinatura.
Result	O terceiro tipo genérico é chamado de Result, e o mesmo objeto que retorna do método <code>doInBackground()</code> é passado como parâmetro para o método <code>onPostExecute()</code> . O método <code>onPostExecute()</code> executa na UI Thread e pode atualizar a interface. No exemplo que criamos, o retorno do método <code>doInBackground()</code> era do tipo <code>Long</code> , e o método <code>onPostExecute(Long result)</code> recebe um <code>Long</code> como argumento.

Eu sei, parece complicado, mas logo você se acostuma com a ideia. Mas isso nada mais é do que o conceito de Generics da linguagem Java. Qualquer dúvida, continue lendo e depois volte aqui para revisar os conceitos, não se preocupe se não entender exatamente o que significa cada parâmetro agora.

Outra informação importante é que, ao criar a `AsyncTask`, você pode definir os tipos genéricos que quiser e deixar alguns como `Void`. O próximo exemplo mostra como deixar todos os argumentos como `Void` e somente o argumento `Result` como `Boolean`. Veja como fica a sintaxe:

```
private class DownloadBitmapTask extends AsyncTask<Void, Void, Boolean> {
    Bitmap bitmap;
    @Override
    protected Boolean doInBackground(Void... params) {
        bitmap = downloadBitmap();
        return true;
    }
    protected void onPostExecute(Boolean ok) {
        if(ok) {
            // atualizar a view aqui :-)
        }
    }
}
```

Às vezes, um código e um exemplo falam mais que mil palavras. Vamos criar novamente o exemplo do download da imagem, mas desta vez utilizando a classe AsyncTask:



MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    private static final String URL = "http://livroandroid.com.br/imgs/livro_android.png";
    private ProgressBar progress;
    private ImageView imgView;
    private Bitmap bitmap;
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_download_imagem);
        imgView = (ImageView) findViewById(R.id.img);
        progress = (ProgressBar) findViewById(R.id.progress);
        // Faz o download
        downloadImagem();
    }
    // Faz o download da imagem em uma nova thread
    private void downloadImagem() {
        // Cria uma AsyncTask
        DownloadTask task = new DownloadTask();
        // Executa a task/thread
        task.execute();
    }
    private class DownloadTask extends AsyncTask<Void,Void,Bitmap> {
        @Override
        protected void onPreExecute() {
            super.onPreExecute();
            // Executa antes do download. Mostra o ProgressBar para fazer a animação.
            progress.setVisibility(View.VISIBLE);
        }
        @Override
        protected Bitmap doInBackground(Void... params) {
            // Faz o download em uma thread e retorna o bitmap.
            try {
                bitmap = Download.downloadBitmap(URL);
            } catch (Exception e) {
                Log.e("livroandroid",e.getMessage(), e);
            }
            return bitmap;
        }
    }
}
```

```

protected void onPostExecute(Bitmap bitmap) {
    if(bitmap != null) {
        // Atualiza a imagem na UI Thread
        imageView.setImageBitmap(bitmap);
        // Esconde o progress
        progress.setVisibility(View.INVISIBLE);
    }
}
}
public Context getContext() { return this; }
}

```

O layout da activity é o mesmo do exemplo anterior do download da imagem. E, ao executar o código, o resultado também será o mesmo.

O importante é você perceber que a `AsyncTask` encapsulou a necessidade de manipular as threads e handlers, além de trazer outros benefícios os quais não utilizei neste exemplo, como cancelar as tarefas ou utilizar o pool de threads para executar o processo de forma serial ou paralela.

Nota: uma das vantagens de utilizar a classe `AsyncTask` é que é possível parar a execução da tarefa com o método `cancell(boolean)`. Quando uma tarefa é cancelada, o método `onCancelled()` é chamado. Para descobrir se uma tarefa já foi cancelada, basta chamar o método `isCancelled()`. É comum nos métodos `onPause()` da activity e fragment chamar o método `cancell(boolean)` da `AsyncTask`; assim, quando a activity vai entrar em estado de pausa, essas tarefas são interrompidas para economizar recursos, uma vez que a tela será fechada. Tudo depende do caso e se você deseja ou não parar a execução da tarefa.

10.8 Download de imagens com a biblioteca Picasso

O principal objetivo deste capítulo foi lhe explicar a importância do uso de threads no aplicativo e consequentemente explicamos as classes `Handler` e `AsyncTask`. Como o exemplo de threads que criamos fez o download de uma imagem, vamos aproveitar e mostrar uma biblioteca que facilita este download, chamada de **Picasso** (<http://square.github.io/picasso/>).

Essa biblioteca é utilizada por muitos desenvolvedores, pois é extremamente simples e inclusive faz cache das imagens. Para utilizá-la, basta declarar a seguinte dependência no arquivo `app/build.gradle`.

 /app/build.gradle

```
dependencies {  
    . . .  
    compile 'com.squareup.picasso:picasso:2.5.2'  
}
```

Depois de declarar a dependência, o download pode ser feito com apenas uma única linha de código.

```
Picasso.with(this).load(URL).into(ingView);
```

Também é possível especificar uma imagem que deve ser exibida antes de o download terminar (placeholder) é uma imagem de erro.

```
Picasso.with(this).load(URL).placeholder(R.drawable.android).error(R.drawable.android)  
    .into(ingView);
```

Caso você queira animar um `ProgressBar` como fizemos nos exemplos anteriores, basta mostrar o progresso antes do download, e escondê-lo quando o download terminar. A biblioteca Picasso oferece a interface de `Callback` para avisar o aplicativo sobre o resultado do download, seja sucesso ou falha.

```
progress.setVisibility(View.VISIBLE);  
Picasso.with(this).load(URL)  
    .placeholder(R.drawable.android)  
    .error(R.drawable.android)  
    .into(ingView, new Callback() {  
        @Override  
        public void onSuccess() { // OK  
            progress.setVisibility(View.GONE);  
        }  
        @Override  
        public void onError() { // ERRO  
            progress.setVisibility(View.GONE);  
        }  
    });
```

É isso: simples e prático.

Dica: mostrei como animar um `ProgressBar` para seu aprendizado, e isso é muito utilizado em várias situações. Porém, no caso de download de imagens, parece que está havendo uma tendência de utilizar apenas uma imagem temporária durante o download, chamada de placeholder. Um exemplo é a navegação no aplicativo do Google Play que mostra uma imagem vazia enquanto o download

do ícone do aplicativo não termina. Nesse contexto, a biblioteca Picasso é extremamente útil, pois faz o download em uma única linha de código.

10.9 Links úteis

Neste capítulo, estudamos a classe `Handler`, que tem como principal objetivo enviar uma mensagem para ser processada na UI Thread. Também estudamos a importância de criar threads e por último aprendemos a utilizar a classe `AsyncTask`.

No que se refere principalmente à classe `AsyncTask`, ainda temos muito o que estudar. Como eu expliquei anteriormente, a classe `AsyncTask` é uma biblioteca de threads e contém vários recursos interessantes. Acredito que esta introdução sobre threads e `AsyncTask` já é suficiente para avançarmos com nossos estudos.

Eu quero entrar logo nos capítulos 11 e 12, sobre Material Design e Toolbar respectivamente, para logo depois começar o desenvolvimento do aplicativo dos carros passo a passo. Nesse aplicativo também vamos usar threads, pois os carros serão buscados de um web service, portanto vamos aprender na prática tudo o que for necessário. No final do livro, deixei um capítulo especial mais avançado sobre a classe `AsyncTask` (Capítulo 31), que acredito que agora não seja o momento certo para ler. Vamos dar um passo de cada vez.

Para complementar seus estudos, separei alguns links da documentação oficial.

- **Communicating with the UI Thread**

<https://developer.android.com/training/multiple-threads/communicate-ui.html>

- **Processes and Threads**

<http://developer.android.com/guide/components/processes-and-threads.html>

- **Picasso**

<http://square.github.io/picasso>



CAPÍTULO 11

Material Design

Com a chegada do Android 5.0 (Lollipop), foi criado o Material Design, um guia completo sobre como criar aplicações com um ótimo design e que leva em consideração que atualmente o Android está difundido em diversos dispositivos como smartphones, tablets, wearables, óculos, TVs e carros.

Com esses avanços da plataforma, foi necessário criar um guia de Design, e principalmente uma interface que funcione de forma consistente, independentemente da plataforma e do tipo do dispositivo, seja um pequeno relógio ou uma grande TV.

11.1 Introdução

O Material Design é um guia completo sobre como implementar o visual, animações e interação entre os componentes de um layout, considerando que o Android se tornou uma plataforma comum para vários dispositivos, como smartphones e tablets (Android), wearables (Android Wear), óculos (Google Glass), TVs (Android TV) e carros (Android Auto).

Implementar o visual de um aplicativo de forma consistente, simples e intuitiva para cada tipo de dispositivo é um desafio, e o Material Design é o resultado do esforço do Google de padronizar um guia completo de design para nos auxiliar nessa tarefa.

No Android 5.0 (Lollipop), foram criadas diversas APIs para auxiliar o desenvolver a criar interfaces ricas, fluidas e com animações iguais àquelas encontradas nos aplicativos nativos do Google. O melhor de tudo é que podemos utilizar uma biblioteca de compatibilidade para trazer os benefícios do Material Design, inclusive para dispositivos com versões antigas do Android.

A figura 11.1 é da documentação oficial do Android e mostra a ideia do Material Design: um aplicativo que tem um design consistente em diversos tipos de dispositivos.



Figura 11.1 – Material Design.

11.2 Tema Material

No capítulo 5, sobre action bar, estudamos a importância dos temas na plataforma de desenvolvimento do Android. Vimos que o tema Holo revolucionou o desenvolvimento de aplicativos, pois introduziu a action bar, e o tema Material foi criado junto com o Android 5.0 (Lollipop).

Nesta altura do livro, você já sabe configurar a biblioteca de compatibilidade no projeto e também sabe utilizar o tema Material. Conforme estudamos, basta configurar o tema do aplicativo para herdar de algum destes temas:

- `android:/Theme.Material`
- `android:/Theme.Material.Light`
- `android:/Theme.Material.Light.DarkActionBar`

Caso o aplicativo utilize a action bar de compatibilidade, deve-se utilizar o tema `AppCompat`, que é consistente com todas as versões do Android.

- `Theme.AppCompat`
- `Theme.AppCompat.Light`
- `Theme.AppCompat.Light.DarkActionBar`

Lembre-se de que, para utilizar o tema `AppCompat`, você deve declarar no arquivo `app/build.gradle` a dependência para a biblioteca `support:appcompat-v7` e todas as activities devem herdar de `AppCompatActivity`.

```
app/build.gradle

dependencies {
    . . .
    compile 'com.android.support:appcompat-v7:22.1.0'
}
```

Nota: para um melhor entendimento dos próximos exemplos, recomendo abrir o projeto de exemplo deste capítulo no Android Studio e acompanhar a explicação.

11.3 Paleta de cores

O Material Design utiliza muito o conceito de ter uma cor padrão (primary color) para o aplicativo e uma cor de acentuação (accent color) para dar destaque a algumas views. No capítulo 7, sobre a classe `View`, já expliquei um pouco sobre esse assunto, e agora vamos estudá-lo um pouco mais a fundo.

O mais legal do tema Material é que o Google tornou bem simples customizar as cores do tema, algo que sempre foi uma dor de cabeça nos temas antigos. Para customizar as cores, basta sobrescrever algumas propriedades do tema, conforme indicado pela figura 11.2.

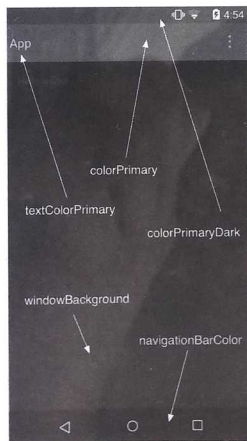


Figura 11.2 – Cores para customizar o tema Material.

Para brincarmos, crie o projeto `HelloMaterial`, ou abra o projeto de exemplo deste capítulo. Ao criar o projeto com o Android Studio, deixe ele compatível com o Android 2.3 ou superior e ative a biblioteca de compatibilidade `appcompat-v7` no arquivo `app/build.gradle`. Lembre-se também de que a `MainActivity` deve ser filha de `AppCompatActivity`.

Para começar, vamos customizar as cores do tema. O que eu costumo fazer é customizar a cor primária e a cor de acentuação, conforme demonstrado a seguir. Note que, como estamos utilizando o tema `AppCompat`, só precisamos de uma versão do arquivo `/res/values/styles.xml`.



`/res/values/styles.xml`

```
<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Cor primária do aplicativo -->
        <item name="colorPrimary">@color/primary</item>
        <!-- Variação escura da cor primária para a status bar e contextual app bars -->
        <item name="colorPrimaryDark">@color/primary_dark</item>
        <!-- Cor de acentuação para as views (checkbox, textfield etc.) -->
        <item name="colorAccent">@color/accent</item>
    </style>
</resources>
```

Estou customizando as cores no arquivo do tema, e por isso criei estas cores no arquivo `/res/values/colors.xml`.



`/res/values/colors.xml`

```
<resources>
    <!-- Cores do tema Material -->
    <color name="primary">#03A9F4</color>
    <color name="primary_dark">#01579B</color>
    <color name="accent">#F44336</color>
</resources>
```

Simples assim. Se você executar o projeto com essas configurações, a cor da action bar será azul, a cor da status bar será um azul escuro, e a cor de acentuação utilizada para dar destaque é vermelho.

Nota: a cor primária (`primary`) deve ser a cor principal do aplicativo. A cor primária escura (`primary escura`) é uma variação escura da cor primária e é utilizada na status bar. A cor de acentuação (`accent color`) é de extrema importância para destacar views importantes do aplicativo e chamar a atenção do usuário.

Para escolher essas cores, o Google recomenda seguir a paleta de cores, que pode ser encontrada nesta página da documentação oficial:

<http://www.google.com/design/spec/style/color.html>

Você vai encontrar diversas cores nessa página, e todas elas terão vários níveis de intensidade. Segundo as boas práticas do Material Design, a cor primária (primary) do aplicativo deve ser alguma cor com intensidade 500 e a cor primária forte (primary dark) deve ter intensidade 700 ou superior.

Para entender do que estou falando, abra o link mencionado anteriormente e veja a paleta de cores.

11.4 Elevação de views

No Material Design, as views podem ser elevadas da superfície, aumentando ou diminuindo a sombra sobre elas. Para alterar a elevação de uma view no XML, utilize o atributo `android:elevation` ou utilize o método `setElevation(z)` para alterar pelo código.

A figura 11.3 demonstra o conceito de elevação. Na esquerda podemos visualizar a view normal. Na direita podemos visualizar o resultado se aplicada a elevação no eixo Z da view.

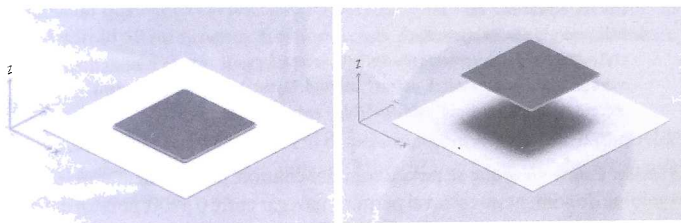


Figura 11.3 – Elevação e sombra da view.

Esse conceito é um dos pilares do Material Design, pois os elementos visuais são construídos e aplicados sobre camadas que se sobrepõem. A figura 11.4 demonstra a ideia. Na esquerda as views estão sobrepostas (correto) e na direita não temos ideia de profundidade (incorreto).

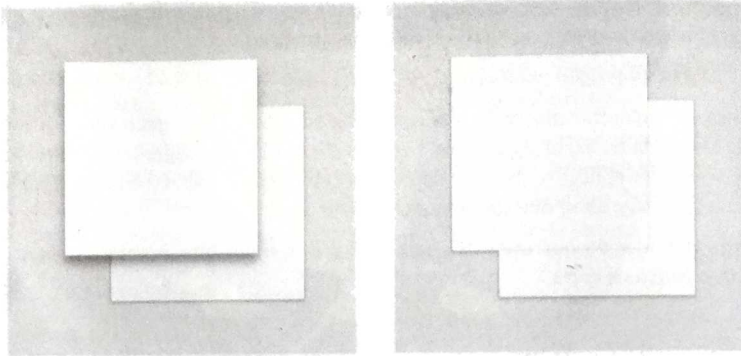


Figura 11.4 – Elevação e sombra da view.

Para praticarmos o conceito de elevação, vamos criar um exemplo. O arquivo de layout contém um botão cuja elevação vamos alterar via programação.

 /res/layout/activity_exemplo_elevation.xml

```
<LinearLayout . . .>
    <Button android:id="@+id/button" android:text="@string/hello_world"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:elevation="2dp" />
    <SeekBar android:id="@+id/seekBar"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:max="100" android:padding="20dp" android:progress="0" />
</LinearLayout>
```

A `SeekBar` é uma view que se parece com um controle para aumentar ou diminuir o volume do som; neste caso, vai permitir navegar entre 0 a 100, pois foi definida a propriedade `android:max="100"`. No código da activity, sempre que o valor da `SeekBar` for alterado, vamos configurar esse mesmo valor na elevação do botão.

 `ExemploElevationActivity.java`

```
public class ExemploElevationActivity extends AppCompatActivity {
    private SeekBar seekBar;
    private Button button;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

```

setContentView(R.layout.activity_exemplo_elevation);
button = (Button) findViewById(R.id.button);
getSupportActionBar().setDisplayHomeAsUpEnabled(true);
seekBar = (SeekBar) findViewById(R.id.seekBar);
seekBar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
    @Override
    public void onProgressChanged(SeekBar seekBar, int progress, boolean fromUser) {
        // Altera a elevação da view ao mexer na SeekBar
        button.setElevation(progress);
    }
    @Override
    public void onStartTrackingTouch(SeekBar seekBar) { }
    @Override
    public void onStopTrackingTouch(SeekBar seekBar) { }
});
}
}

```

Para entender esse exemplo, você precisa executar no emulador. Ao mexer na Seekbar, a elevação da view será alterada e você perceberá o efeito que acontece na sombra da view. É como se a view estivesse levantando e saindo da tela. A figura 11.5 mostra o valor 0 (zero) configurado como elevação e depois o valor 100. Veja que no valor 100 a sombra da view é grande.

Vale ressaltar que o método `setElevation(float)` da classe `View` está disponível apenas para o Android 5.0 ou superior. Em nosso caso, por questões de compatibilidade, pode ser utilizada a classe `ViewCompat`, conforme demonstrado a seguir:

```
ViewCompat.setElevation(view, valor);
```

A classe `ViewCompat` aplica a elevação em dispositivos com Android 5.0 ou superior, ou simula a elevação em versões antigas. O importante é que o código fica compatível com todas as versões.



Figura 11.5 – Exemplo de elevação e sombra da view.

Importante: execute esses exemplos no emulador do Android 5.0 ou superior para obter os efeitos desejados do tema Material. Por questões de compatibilidade com versões antigas do Android, recomendo utilizar a classe `ViewCompat`.

11.5 Ripple – feedback ao toque

Ripple é o nome do efeito de toque padrão do Android 5.0, e todas as views como botões, listas e ações da action bar o têm por padrão. Esse efeito de toque no Android é conhecido como Touch Feedback (feedback ao toque), e seu objetivo é informar ao usuário que o sistema recebeu o toque de uma forma leve e agradável.

O efeito de ripple é alcançado pela classe `RippleDrawable`, que pode ser criada por programação, ou até mesmo via XML criando uma tag `<ripple>`. Neste próximo exemplo, vamos testar em um botão vários efeitos diferentes. Para isso abra o arquivo `/res/layout/activity_exemplo_ripple.xml` no editor visual (Figura 11.6).

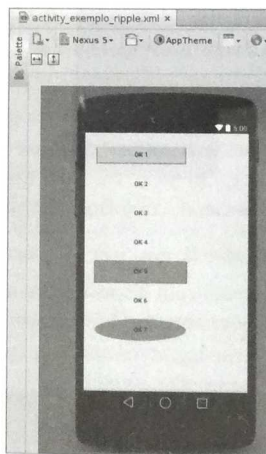


Figura 11.6 – Efeitos ao clicar em um botão.

Cada botão está configurado com um efeito diferente, mas para entender a explicação, por favor, execute o projeto no emulador para visualizar o efeito. Não tem como demonstrar as animações no livro.

Agora que você já executou o exemplo, vamos continuar com a explicação.

O primeiro botão não faz nada, mas caso o aplicativo seja executado em um Android 5.0 ou superior o efeito de ripple é criado automaticamente.

Para dar esse feedback ao toque e adicionar o efeito de ripple e qualquer view, podemos utilizar o item `?attr/selectableItemBackground`, que deve ser aplicado como fundo (background) de uma view. O segundo botão mostrou justamente isso.

```
<!-- Botão 2: Efeito ripple -->
<Button android:text="@string/ok2" . . .
    android:background="?attr/selectableItemBackground" />
```

A figura 11.7 demonstra a animação criada ao tocar no botão. A primeira parte mostra o toque, e depois o efeito da onda foi crescendo e se propagando até ocupar o botão inteiro. O item `?attr/selectableItemBackground` é muito utilizado ao construir uma lista com `ListView`, com o objetivo de configurar o efeito de toque da lista.

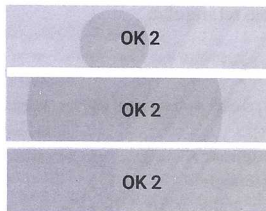


Figura 11.7 – Feedback ao toque.

Nota: a animação de ripple é parecida com o efeito de quando você toca alguma superfície com água, formando aquelas ondas circulares.

O terceiro botão mostrou o efeito criado pelo item `?android:attr/selectableItemBackgroundBorderless`, o qual faz o efeito da onda porém sem margens.

```
<!-- Botão 3: Efeito ripple sem borda -->
<Button android:text="@string/ok3" . . .
    android:background="?attr/selectableItemBackgroundBorderless" />
```

É como se o efeito continuasse executando, pois não bateu em nenhuma borda para parar. A figura 11.8 demonstra que o efeito sem borda inclusive invadiu o espaço dos outros botões.

Os próximos botões, 4, 5, 6 e 7, mostram como criar um efeito de ripple customizado com um arquivo XML; assim podemos definir inclusive as cores do botão e do efeito da onda.

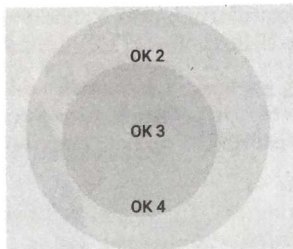


Figura 11.8 – Feedback ao toque com o efeito da onda sem a borda.

Um ripple é criado com a tag <ripple> e o atributo android:color define a cor do efeito das ondas. A tag <shape> define o formato da borda, que pode ser retangular (shape="rectangle") ou oval (shape="oval"). A seguir, temos um exemplo de arquivo que cria um efeito de ripple retangular.

```

<?xml version="1.0" encoding="utf-8" />
<resources>
    <ripple xmlns:android="http://schemas.android.com/apk/res/android"
        android:color="@color/accent">
        <item android:id="@android:id/mask">
            <shape android:shape="rectangle">
                <solid android:color="@color/primary" />
            </shape>
        </item>
    </ripple>
</resources>

```

Esse arquivo customizado pode ser definido como o fundo de qualquer view, conforme demonstrado a seguir.

```

<!-- Botão 4: Efeito ripple (quadrado sem preencher) -->
<Button android:text="@string/ok4" . . .
    android:background="@drawable/ripple_rect" />

```

Para conferir os resultados, execute o projeto de exemplo deste capítulo.

Nota: segundo as boas práticas do Material Design, é importante fornecer o feedback ao toque (Touch Feedback) para interagir com o usuário. Esses conceitos podem ser aplicados em qualquer tipo de view.

11.6 Floating Action Button (FAB)

Um dos novos padrões mais utilizados do Material Design é o **Floating Action Button**, que consiste em um botão flutuante que deve ser utilizado para conter a ação mais importante da tela. Esse botão deve ter uma cor chamativa e por isso é recomendado utilizar a cor de acentuação definida no tema.

Vale ressaltar que o **Floating Action Button** é frequentemente chamado apenas de FAB, portanto acostume-se com essa sigla. Felizmente, criar um FAB é bem simples e você pode utilizar um `ImageButton` com um fundo especial, conforme o código a seguir:

```

<?xml version="1.0" encoding="utf-8" ?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="16dp">
    <TextView android:text="@string/hello_world"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <ImageButton android:id="@+id/btOk"
        android:layout_width="56dp" android:layout_height="56dp"
        android:layout_gravity="center"
        android:src="@android:drawable/ic_input_add"
        android:tint="#fff000"
        android:background="@drawable/fab_oval"
        android:elevation="8dp" />
</RelativeLayout>

```

Esse botão foi configurado com uma imagem nativa do Android, sendo que a notação `@android:drawable/nome_imagem` é utilizada para acessar um recurso nativo do Android. Em nosso exemplo, estamos utilizando a imagem `ic_input_add`. Se você abrir essa figura com a tecla **Ctrl+clique** no editor, verá que ela é um sinal de mais (+) transparente, conforme a figura 11.9.



Figura 11.9 – Imagem transparente.

O segredo dessa imagem é que a área do sinal de mais (+) pode ser preenchida com qualquer cor, e por isso é definido o atributo `android:tint="#fff000"` que está pintando a cruz de amarelo. Já o fundo do botão é feito com um efeito de ripple, sendo assim basta criar um XML customizado conforme demonstrado a seguir:

```
 /res/drawable/fab_oval.xml  
  
<?xml version="1.0" encoding="utf-8"?>  
<ripple xmlns:android="http://schemas.android.com/apk/res/android"  
    android:color="?android:colorPrimary">  
    <item>  
        <shape android:shape="oval">  
            <solid android:color="?android:colorAccent" />  
        </shape>  
    </item>  
</ripple>
```

É isso é tudo! Neste exemplo, a figura `ic_input_add` mostra o sinal de mais (+), que é padrão do Android, e o fundo do ripple criou um `shape=oval`, por isso a imagem ficou como um círculo. O fundo do ripple foi definido como vermelho, que é a cor de acentuação configurada no projeto. A figura 11.10 mostra o resultado, e quando você executar no emulador verá que o botão tem o Touch Feedback com o efeito do ripple.

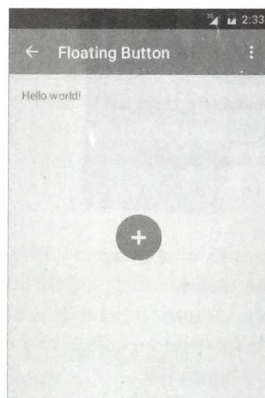


Figura 11.10 – Floating Action Button.

Nota: a imagem do sinal de mais (+) transparente é conhecida como *Tint Drawable Resources*. No Android 5.0 (API Level 21) ou superior é possível criar as figuras como *nine-patches* e definir uma máscara transparente (alpha masks). Essa área transparente pode ser pintada (tint) com a cor desejada.

Esse exemplo que fizemos sobre o botão FAB é interessante porque você aprendeu conceitos importantes como *Tint Drawable Resources* e novamente utilizamos

o efeito de ripple. Porém, logo depois do Google I/O 2015, o Google anunciou a biblioteca *Android Design Support Library* que contém classes e componentes para auxiliar na construção de aplicativos com Material Design. Dentre esses componentes, foi criada a classe `FloatingActionButton`, que é uma subclasse de `ImageView` e torna simples a tarefa de criar um **Floating Action Button**. Para utilizar a biblioteca *Android Design Support Library*, declare a dependência no *app/build.gradle*.

 **app/build.gradle**

```
...  
compile 'com.android.support:design:22.2.0'
```

Utilizando esta biblioteca, criar um *Floating Action Button* é um passe de mágica, conforme mostra o código a seguir. Vale ressaltar, que por padrão a cor do fundo do botão utiliza a cor de acentuação definida no tema material.

```
<android.support.design.widget.FloatingActionButton  
    android:id="@+id/fab"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_gravity="center"  
    android:src="@android:drawable/ic_input_add"  
    android:tint="#fff000"  
>
```

11.7 CardView

O Material Design recomenda a utilização de cartões (cards) sempre que for preciso separar a visualização de determinados elementos. O visual é parecido com o que temos na lista do Google Now, em que diversos cartões são mostrados em uma lista. Cartões também podem ajudar a organizar o conteúdo de aplicativos que mostram informações em listas ou grids, como é feito no Google Play, oferecendo uma melhor visualização ao usuário.

A figura 11.11 mostra o visual obtido com os cartões no aplicativo do Google Play.

A principal ideia por trás da utilização dos cartões é proporcionar uma interface consistente em todas as plataformas do Android, desde smartphones, tablets e até relógios. Sendo assim, criar um cartão (card) é a tarefa mais simples, o importante é você entender o real significado de utilizá-lo, que é seguir as boas práticas de interface do Material Design e oferecer um design de interface consistente para vários tipos de dispositivos.

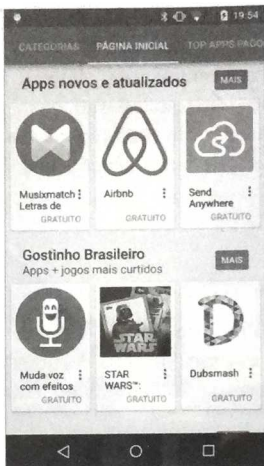


Figura 11.11 – Google Play.

Agora vamos falar um pouco de código. A classe do `CardView` é distribuída por meio de uma biblioteca de compatibilidade do Google, portanto declare a seguinte dependência no projeto:

```

app/build.gradle

dependencies {
    ...
    compile 'com.android.support:cardview-v7:21.0.+'
}

```

A classe `CardView` é filha de `FrameLayout` e pode ser utilizada como um contêiner de outras views. Utilizando o `CardView` podemos criar uma borda quadrada ou circular ao redor do layout. Neste próximo exemplo eu adicionei o `CardView` no layout e dentro dele coloquei um simples `TextView`, conforme podemos visualizar no código a seguir.

```

/res/layout/activity_exemplo_card_view.xml

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="16dp" android:orientation="vertical">

```

```

<android.support.v7.widget.CardView
    android:id="@+id/cardView"
    android:layout_width="100dp" android:layout_height="100dp"
    card_view:contentPadding="6dp"
    card_view:cardBackgroundColor="?attr/colorPrimary">
    <TextView
        android:text="@string/hello_world"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:textColor="?attr/colorAccent" />
</android.support.v7.widget.CardView>
<SeekBar android:id="@+id/seekBar1"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="fill_parent" android:layout_height="wrap_content"
    android:max="100" android:padding="20dp" android:progress="0" />
<SeekBar android:id="@+id/seekBar2"
    style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="fill_parent" android:layout_height="wrap_content"
    android:max="100" android:padding="20dp" android:progress="0" />
</LinearLayout>

```

A propriedade `card_view:cardBackgroundColor` define a cor de fundo do cartão, e a propriedade `card_view:contentPadding` define o espaçamento do seu conteúdo. Também pode ser definida a propriedade `card_view:cardCornerRadius` para deixar a borda do cartão arredondada, pois o padrão é retangular. E por último podemos alterar a elevação do cartão com a tag `card_view:cardElevation`.

Para você entender como as propriedades `cardCornerRadius` e `cardElevation` afetam o visual do cartão, eu adicionei duas views do tipo `SeekBar` no layout. A primeira `seekBar` vai alterar a elevação do cartão, e a segunda vai aumentar o arredondamento das bordas, conforme o código-fonte demonstrado a seguir.



ExemploCardViewActivity.java

```

public class ExemploCardViewActivity extends AppCompatActivity implements SeekBar.
OnSeekBarChangeListener {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_exemplo_card_view);
        cardView = (CardView) findViewById(R.id.cardView);
        . . .
    }
    @Override

```



```
public void onProgressChanged(SeekBar seekBar, int progress, boolean fromUser) {  
    if (seekBar == this.seekBar1) {  
        cardView.setCardElevation(progress); // Elevação  
    } else if (seekBar == this.seekBar2) {  
        cardView.setRadius(progress); // Arredondamento  
    }  
}
```

A figura 11.12 mostra o resultado. Veja que como o segundo `SeekBar` está com o valor cheio, a borda do cartão ficou arredondada. Sugiro que você execute o código no emulador para visualizar o resultado.

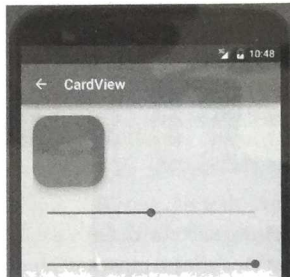


Figura 11.12 – CardView.

Nota: o `CardView` é muito utilizado nos aplicativos nativos do Android. Se você notar, o Google Now é composto por cards e o Google Play também mostra a lista de aplicativos com cards. Esse efeito de cartão ajuda a separar uma view da outra, pois cria essas bordas. No aplicativo dos carros que logo vamos começar a desenvolver vamos utilizar o `CardView` para separar as informações de um carro para outro.

11.8 RecyclerView

Desde a primeira versão do Android, o `ListView` sempre foi a view padrão para criar listas, mas sempre que era necessário criar efeitos customizados ele tornava a vida do desenvolvedor um pouco difícil. Outra questão importante é que se o desenvolvedor não soubesse otimizar a rolagem da lista, por meio do padrão `ViewHolder`, o aplicativo acabava ficando com a rolagem da lista prejudicada, de uma forma não fluida e com travamentos.

Com o surgimento do Material Design, também foi criado o RecyclerView, que é o novo ListView do Android, e a partir de agora é a view recomendada para criar listas segundo as boas práticas de interface.

O RecyclerView apresenta algumas funcionalidades interessantes:

- Suporte a animações ao adicionar ou remover elementos da lista.
- Controle automático da reutilização das views (padrão ViewHolder).
- Permite alterar o gerenciador de layout para renderizar as views como listas, grids, ou outra forma customizada.

Para usar um RecyclerView, basta incluí-lo no layout e informar uma subclasse de RecyclerView.LayoutManager, conforme demonstrado a seguir.

```
RecyclerView recyclerView = (RecyclerView) view.findViewById(R.id.recyclerView);
RecyclerView.LayoutManager layoutManager = new LinearLayoutManager(getActivity());
recyclerView.setLayoutManager(layoutManager);
recyclerView.setItemAnimator(new DefaultItemAnimator());
```

O RecyclerView também utiliza o conceito de adapters para preencher o conteúdo da lista, sendo que um adapter deve ser uma subclasse de RecyclerView.Adapter.

```
List<Planeta> planetas = Planeta.getPlanetas();
recyclerView.setAdapter(new PlanetaAdapter(this, planetas, onClickPlaneta()));
```

O conceito é simples e segue o mesmo princípio de outras views que usam o adapter. Mas o RecyclerView tem uma configuração diferente, que é o gerenciador de layout, chamado de LayoutManager. Basicamente o Google separou as responsabilidades de quem faz o controle do reaproveitamento das views e de quem organiza o layout. Por exemplo, o ListView faz o scroll das suas views e ainda faz um gerenciamento para reaproveitá-las, e para isso usamos o padrão ViewHolder.

Nota: o padrão ViewHolder sempre foi utilizado para fazer a rolagem de um ListView. Não cheguei a explicá-lo aqui no livro porque vamos utilizar o RecyclerView daqui para frente, e ele já faz a rolagem de forma eficiente. Mas vamos explicar resumidamente o que é o ViewHolder. Quando o Android faz a rolagem em uma lista com uma grande quantidade de elementos, é preciso reutilizar as views para evitar criar uma nova a cada item. Imagine que exista uma lista com 1.000 linhas. Como o Android mostra no máximo umas 10 views por vez, podemos criar apenas 10 views e reutilizá-las ao fazer a rolagem, em vez de criar 1.000 objetos do tipo view. Isso otimiza a memória e deixa a rolagem da lista fluida. Com a utilização do RecyclerView, a implementação desse padrão é feita de forma transparente ao desenvolvedor.

O RecyclerView na verdade é um componente especializado no reaproveitamento das views, ou seja, ele recicla as views e implementa automaticamente o padrão ViewHolder, para garantir um bom desempenho ao fazer rolagem com uma grande quantidade de itens. Mas o RecyclerView não sabe desenhar nada na tela, e para isso ele precisa de alguma subclasse de LayoutManager, a qual é responsável por desenhar e organizar a disposição das views. A lista a seguir mostra algumas das subclasses de RecyclerView.LayoutManager:

LinearLayoutManager

Organiza as views na vertical ou horizontal. Para ter o mesmo comportamento do ListView, basta utilizar este gerenciador de layout.

GridLayoutManager

Organiza as views em um grid.

StaggeredGridLayoutManager

Organiza as views em um grid que suporta as orientações vertical e horizontal.

Outra melhoria do RecyclerView é no suporte as animações. Você pode informar uma subclasse de RecyclerView.ItemAnimator que é responsável por animar a lista quando os dados são alterados, como, por exemplo, quando um item é removido ou adicionado.

```
recyclerView.setItemAnimator(new DefaultItemAnimator());
```

A classe DefaultItemAnimator é filha de RecyclerView.ItemAnimator e implementa as animações básicas quando um item da lista é adicionado, removido ou movido de posição. Depois de inserir alguma informação na lista que é a fonte do conteúdo do adapter você pode chamar o método notifyItemInserted(idx) para informar que um item foi adicionado ou o método notifyItemRemoved(idx) para informar que um item foi removido da posição indicada.

Depois dessa explicação, vamos para a parte prática. Primeiramente, para utilizar o RecyclerView é preciso declarar a seguinte dependência:

 app/build.gradle

```
dependencies {  
    ...  
    compile 'com.android.support:recyclerview-v7:21.0.+'  
}
```

A seguir podemos visualizar o arquivo de layout com o RecyclerView.

```

/res/layout/activity_exemplo_recycler_view.xml

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <android.support.v7.widget.RecyclerView
        android:id="@+id/recyclerView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

O código-fonte a seguir mostra o que a activity precisa fazer, que é configurar o adapter do RecyclerView e o gerenciador de layout. Em nosso exemplo, estou utilizando um `LinearLayoutManager` para criar uma lista. Mas ao executar o exemplo no emulador, você verá dois botões na action bar para alternar entre o modo de visualização por lista e grid, assim podemos brincar um pouco com o RecyclerView.

ExemploRecyclerViewActivity.java

```

...
import android.support.v7.widget.DefaultItemAnimator;
import android.support.v7.widget.GridLayoutManager;
import android.support.v7.widget.LinearLayoutManager;
import android.support.v7.widget.RecyclerView;

public class ExemploRecyclerViewActivity extends AppCompatActivity {
    private RecyclerView recyclerView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_exemplo_recycler_view);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        // RecyclerView
        recyclerView = (RecyclerView) findViewById(R.id.recyclerView);
        recyclerView.setLayoutManager(new LinearLayoutManager(this));
        recyclerView.setItemAnimator(new DefaultItemAnimator());
        recyclerView.setHasFixedSize(true);
        // Planetas e Adapter
        List<Planeta> planetas = Planeta.getPlanetas();
        recyclerView.setAdapter(new PlanetaAdapter(this, planetas, onClickPlaneta()));
    }

    // onClick Planeta
    private PlanetaAdapter.PlanetaOnClickListener onClickPlaneta() {
        return new PlanetaAdapter.PlanetaOnClickListener() {
```

```

@Override
public void onClickPlaneta(View view, int idx) {
    List<Planeta> planetas = Planeta.getPlanetas();
    Planeta p = planetas.get(idx);
    Toast.makeText(getBaseContext(), "Planeta: " + p.nome, Toast.LENGTH_SHORT).show();
}
};
}
private Activity getActivity() { return this; }
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_exemplo_recycler_view, menu);
    return true;
}
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.action_linear_layout) {
        // Troca o modo de visualização para lista
        recyclerView.setLayoutManager(new LinearLayoutManager(this));
        return true;
    } else if (id == R.id.action_grid_layout) {
        // Troca o modo de visualização para grid
        recyclerView.setLayoutManager(new GridLayoutManager(this, 2));
        return true;
    }
    return super.onOptionsItemSelected(item);
}
}
}

```

Conforme podemos ver no código da activity, utilizar o RecyclerView é simples e não temos muitas novidades. Mas é no código do adapter que está todo o segredo. Um adapter do RecyclerView utiliza o conceito de Generics do Java (tipos genéricos) portanto os métodos `onCreateViewHolder()` e `onBindViewHolder()` recebem o tipo genérico declarado na classe, que neste caso é a classe interna `PlanetaAdapter.PlanetasViewHolder`.

PlanetaAdapter.java

```

...
import android.support.v7.widget.RecyclerView;
// Herda de RecyclerView.Adapter e declara o tipo genérico <PlanetaAdapter.
PlanetasViewHolder>

```

```

public class PlanetaAdapter extends RecyclerView.Adapter<PlanetaAdapter.PlanetasViewHolder> {
    protected static final String TAG = "livroandroid";
    private final List<Planeta> planetas;
    private final Context context;
    private final PlanetaOnClickListener onClickListener;
    public interface PlanetaOnClickListener {
        public void onClickPlaneta(View view, int idx);
    }
    public PlanetaAdapter(Context context, List<Planeta> planetas, PlanetaOnClickListener
        onClickListener) {
        this.context = context;
        this.planetas = planetas;
        this.onClickListener = onClickListener;
    }
    @Override
    public PlanetasViewHolder onCreateViewHolder(ViewGroup viewGroup, int viewType) {
        // Este método cria uma subclasse de RecyclerView.ViewHolder
        // Infla a view do layout
        View view = LayoutInflater.from(context).inflate(R.layout.adapter_planeta,
            viewGroup, false);
        // Cria a classe do ViewHolder
        PlanetasViewHolder holder = new PlanetasViewHolder(view);
        return holder;
    }
    @Override
    public void onBindViewHolder(final PlanetasViewHolder holder, final int position) {
        // Este método recebe o índice do elemento, e atualiza as views que estão
        // dentro do ViewHolder
        Planeta c = planetas.get(position);
        // Atualiza os valores nas views
        holder.tNome.setText(c.nome);
        holder.img.setImageResource(c.img);
        // Click
        if (onClickListener != null) {
            holder.itemView.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    // Chama o listener para informar que clicou no Planeta
                    onClickListener.onClickPlaneta(holder.view, position);
                }
            });
        }
    }
}

```

```

@Override
public int getItemCount() {
    return this.planetas != null ? this.planetas.size() : 0;
}
// Subclasse de RecyclerView.ViewHolder. Contém todas as views.
public static class PlanetasViewHolder extends RecyclerView.ViewHolder {
    public TextView tNome;
    ImageView img;
    ProgressBar progress;
    private View view;
    public PlanetasViewHolder(View view) {
        super(view);
        this.view = view;
        // Cria as views para salvar no ViewHolder
        tNome = (TextView) view.findViewById(R.id.tNome);
        img = (ImageView) view.findViewById(R.id.img);
        progress = (ProgressBar) view.findViewById(R.id.progress);
    }
}
}

```

O resultado do exemplo pode ser visualizado na figura 11.13, que mostra a visualização no formato de lista e grid.

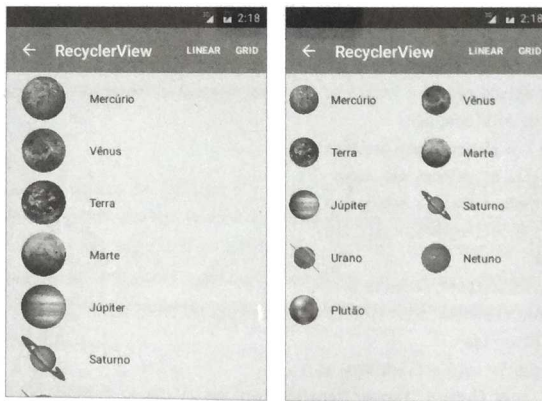


Figura 11.13 – RecyclerView com lista e grid.

Agora vamos entender um pouco o código-fonte desse exemplo, que pode ser um pouco complicado a princípio, mas é simples logo que você se familiarizar. O que

muitos desenvolvedores acham complicado no código é o fato de utilizar classes internas e tipos genéricos, mas isso é Java e não Android.

O RecyclerView basicamente pede que você crie um ViewHolder para armazenar suas views. Isso é feito no método `onCreateViewHolder()`, o qual é chamado uma única vez. Esse ViewHolder é utilizado internamente pelo RecyclerView para fazer o controle e reaproveitamento das views. Para preencher as views com as informações, processo conhecido como "bind", é chamado o método `onBindViewHolder()`, o qual é chamado N vezes conforme a quantidade de elementos da lista.

A classe `PlanetaAdapter` que criamos herda de `RecyclerView.Adapter` e informa no tipo genérico a classe interna `PlanetasViewHolder` criada dentro do adapter. Basicamente temos uma classe interna estática que deve herdar de `RecyclerView.Adapter`.

```
public class PlanetaAdapter extends RecyclerView.Adapter<PlanetaAdapter.  
PlanetasViewHolder> {
```

Como foi declarado na classe o tipo genérico `<PlanetaAdapter.PlanetasViewHolder>`, o método `onCreateViewHolder()` retorna esse mesmo tipo.

```
    public PlanetasViewHolder onCreateViewHolder(ViewGroup viewGroup, int viewType) {  
        // Infla a view do layout  
        View view = LayoutInflater.from(context).inflate(R.layout.adapter_planeta,  
            viewGroup, false);  
        // Cria a classe do ViewHolder  
        PlanetasViewHolder holder = new PlanetasViewHolder(view);  
        return holder;  
    }
```

É neste momento que o layout do adapter é inflado, o qual contém a foto e o nome do planeta. Se quiser, abra o arquivo `/res/layout/adapter_planeta.xml` no editor do Android Studio, pois é um layout simples. O método `onCreateViewHolder()` deve retornar o ViewHolder, para garantir que o padrão ViewHolder é utilizado, com o objetivo de otimizar a rolagem da lista. Com o ListView, esse padrão era utilizado apenas por desenvolvedores que conheciam as boas práticas de desenvolvimento, mas agora ele é automático, pois o RecyclerView exige que você faça isso.

O método `onBindViewHolder()` também recebe o tipo genérico no argumento e deve preencher as views com os valores do objeto.

```
    public void onBindViewHolder(PlanetasViewHolder holder, int position) {  
        // Este método recebe o índice do elemento e atualiza as views.  
        Planeta c = planetas.get(position);  
        // Atualiza os valores nas views  
        holder.tNome.setText(c.nome);
```



```

holder.img.setImageResource(c.img);
// Define o evento de clique, e delega para um listener.
if (onClickListener != null) {
    holder.itemView.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            // Chama o listener para informar que clicou no Planeta
            onClickListener.onClickPlaneta(holder.view, position);
        }
    });
}
}
}

```

O terceiro e último método que precisamos implementar nesse adapter é o `getItemCount()`, o qual retorna a quantidade de linhas.

```

public int getItemCount() {
    return this.planetas != null ? this.planetas.size() : 0;
}

```

Outro conceito importante sobre o `RecyclerView` é que ele não trata os eventos de clique como o `ListView` fazia, justamente por ser um componente especializado no reaproveitamento (`recycle`) de views. Portanto, você precisa criar sua própria interface de listener para tratar esse evento.

Entender isso é pura programação, pois é utilizada uma interface para fazer a comunicação entre dois objetos distintos. Por isso, o adapter define a interface `PlanetaOnClickListener` para a qual ele delega os eventos de clique. A classe da `activity` implementa essa interface e avisa ao `RecyclerView` que ela está interessada em receber os eventos de clique.

```

// Configura o adapter com a lista de planetas e o listener de clique
recyclerView.setAdapter(new PlanetaAdapter(this, planetas, onClickPlaneta()));
// Depois implementa o método onClickPlaneta(view,idx)
private PlanetaAdapter.PlanetaOnClickListener onClickPlaneta() {
    return new PlanetaAdapter.PlanetaOnClickListener() {
        @Override
        public void onClickPlaneta(View view, int idx) {
            // clicou no planeta
        }
    };
}
}

```

Criar esse tipo de interface de retorno é frequentemente conhecido por desenvolvedores de software como interfaces de `callback`, `listener` ou `delegate`.

Dica: outro padrão de design muito conhecido no Material Design é o Swipe to refresh, que permite atualizar os dados da lista quando o usuário faz o gesto de rolagem para baixo. Quando formos desenvolver o aplicativo dos carros, vamos implementar isso.

11.9 Efeito de revelação (Reveal Effect)

No Material Design uma animação muito utilizada é o efeito de revelação, chamado de **Reveal Effect**. Essa animação faz a view aparecer aumentando gradativamente o seu tamanho ou desaparecer diminuindo o seu tamanho.

Para demonstrar como utilizar essa animação, criei a classe `RevealEffect` com os métodos `show()` e `hide()`.



RevealEffect.java

```
@TargetApi(Build.VERSION_CODES.LOLLIPOP)
public class RevealEffect {
    public static void show(View view, long animDuration) {
        // Centro da view
        int cx = (view.getLeft() + view.getRight()) / 2;
        int cy = (view.getTop() + view.getBottom()) / 2;
        // Define o arco para a animação
        int finalRadius = Math.max(view.getWidth(), view.getHeight());
        // Cria a animação
        Animator anim = ViewAnimationUtils.createCircularReveal(view, cx, cy, 0, finalRadius);
        // Inicia a animação
        view.setVisibility(View.VISIBLE);
        anim.setDuration(animDuration);
        anim.start();
    }

    public static void hide(final View view, long animDuration) {
        // Centro da view
        int cx = (view.getLeft() + view.getRight()) / 2;
        int cy = (view.getTop() + view.getBottom()) / 2;
        // Define o arco para a animação
        int initialRadius = view.getWidth();
        // Cria a animação
        Animator anim = ViewAnimationUtils.createCircularReveal(view, cx, cy, initialRadius, 0);
        // Quando a animação terminar esconde a view
        anim.addListener(new AnimatorListenerAdapter() {
```

```

        @Override
        public void onAnimationEnd(Animator animation) {
            super.onAnimationEnd(animation);
            view.setVisibility(View.INVISIBLE);
        }
    });
    // Inicia a animação
    anim.setDuration(animDuration);
    anim.start();
}
}

```

Para testar esse efeito de animação, criei um layout simples com dois botões que chamam os métodos `show()` e `hide()` dessa classe. A imagem do planeta que está no centro do layout vai receber o resultado da animação. A figura 11.14 mostra a animação de revelação executando no emulador, mas o ideal é você conferir no emulador.

No código-fonte da activity, a tarefa necessária é obter a view que precisa ser animada e chamar os métodos para mostrar ou esconder a view.

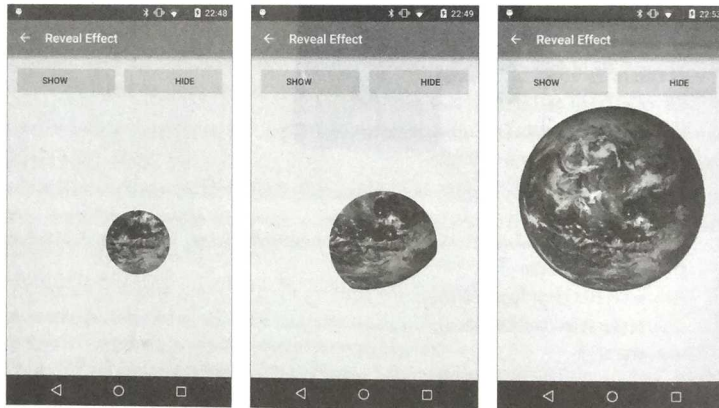


Figura 11.14 – Efeito de revelação.



ExemploRevealEffectActivity.java

```

public class ExemploRevealEffectActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_exemplo_reveal_effect);
    }
}

```

```

        findViewById(R.id.btShow).setOnClickListener(onClickShow());
        findViewById(R.id.btHide).setOnClickListener(onClickHide());
    }
    private View.OnClickListener onClickShow() {
        return new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                View img = findViewById(R.id.img);
                RevealEffect.show(img, 2000);
            }
        };
    }
    private View.OnClickListener onClickHide() {
        return new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                View img = findViewById(R.id.img);
                RevealEffect.hide(img, 2000);
            }
        };
    }
}

```

11.10 Extraindo as cores de uma figura

Como já foi dito, o Material Design é muito baseado em cores. Por isso foi criada a classe `Palette` com o objetivo de extrair as principais cores de uma figura. A ideia de extrair as principais cores da imagem é para colorir o layout e views com as cores da imagem.

Imagens no Android são representadas pela classe `Bitmap`. Assim, para extrair a paleta de cores de uma figura, utilize o seguinte código:

```

Bitmap bitmap = . . .;
Palette p = Palette.generate(bitmap);

```

Preferencialmente devemos utilizar o método `generateAsync(bitmap, listener)` para extrair as cores, pois ele é assíncrono e não vai travar a UI Thread.

```

Palette.generateAsync(bitmap, new Palette.PaletteAsyncListener() {
    public void onGenerated(Palette palette) {
        // Fazer algo com as cores aqui
    }
});

```

Depois que a paleta de cores é extraída da figura, podemos tentar ler as tonalidades de cores que foram extraídas da figura, que são: **Vibrant**, **Vibrant Dark**, **Vibrant Light**, **Muted**, **Muted Dark** e **Muted Light**. Para fazer isso, basta chamar algum método da classe `Palette`, como o método `getVibrantColor(defaultColor)`.

Se quiser brincar um pouco com a paleta de cores, execute no emulador o exemplo que está disponível no projeto deste capítulo. Nele, pinto vários `TextViews` que estão na tela com as cores que são extraídas de uma figura do planeta Terra.

11.11 Animações com item compartilhado entre duas activities

No capítulo 9 sobre animações, aprendemos a criar uma animação ao navegar de uma activity para outra. A boa notícia é que a partir do Android 5.0 (Lollipop) foram criados mais dois métodos que são muito interessantes, pois permitem compartilhar elementos entre duas activities e animar a transição desse elemento compartilhado.

```
makeSceneTransitionAnimation (Activity activity, View sharedElement, String
    sharedElementName)
```

Configura uma animação customizada compartilhando uma view entre duas activities. O parâmetro `sharedElementName` é a chave desta view, que deve existir em ambas as activities. Para utilizar esse método, a activity precisa habilitar a funcionalidade `FEATURE_ACTIVITY_TRANSITIONS`.

```
makeSceneTransitionAnimation (Activity activity, Pair...<View, String>
    sharedElements)
```

Idem ao método anterior, mas permite passar uma lista de views a serem compartilhadas durante a animação de transição.

Nota: o aplicativo do Google Play usa e abusa deste novo recurso. Na página inicial que lista os aplicativos da loja, ao clicar em algum aplicativo, a nova activity é aberta e o ícone do aplicativo move-se até a nova posição.

Para utilizar esse novo recurso de animação, a transição de janelas precisa estar habilitada, o que pode ser feito configurando o *AndroidManifest.xml* ou dinamicamente no código de cada activity.

Basicamente podemos definir a animação de entrada (enter) ao abrir a tela, a animação de saída (exit) ao sair da tela e ainda compartilhar elementos (`sharedElements`) entre as telas.

Para habilitar o modo de transição de telas, temos duas formas. A primeira é configurar o atributo `windowContentTransitions` para `true` na configuração do tema, como demonstrado a seguir.

```
<style name="BaseAppTheme" parent="android:Theme.Material">
    <!-- Habilita a transição de telas -->
    <item name="android:windowContentTransitions">true</item>
    . . .
</style>
```

As animações de transição suportadas no Android 5.0 são:

- **explode** – Move uma view para dentro ou fora da tela.
- **slide** – Move a view para dentro ou fora da tela, fazendo um movimento lateral.
- **fade** – Utiliza a propriedade `alpha` das views para controlar a transparência a fim de fazer a transição.

No arquivo de manifesto, efeitos de transição podem ser configurados de forma global para todo o tema, como por exemplo:

```
<style name="BaseAppTheme" parent="android:Theme.Material">
    <!-- Habilita a transição de telas -->
    <item name="android:windowContentTransitions">true</item>
    . . .
    <!-- Animações de entrada e saída -->
    <item name="android:windowEnterTransition">@transition/explode</item>
    <item name="android:windowExitTransition">@transition/explode</item>
</style>
```

Outra opção é habilitar o modo de transição entre atividades pelo código. Basta chamar o método `getWindow().requestFeature(Window.FEATURE_CONTENT_TRANSITIONS)`. Eu costumo utilizar essa opção. Só tenha atenção, pois para a transição funcionar esta chamada precisa ser feita na primeira linha de código do método `onCreate(bundle)` da activity e deve ser incluída em ambas as activities, origem e destino da animação.

Mas chega de teoria, vamos praticar um pouco de código. Copie o projeto **HelloActivityTransition** que fizemos no capítulo 9, sobre animações, pois ele será a base para continuarmos. No capítulo 9 já aprendemos a fazer transições de transparência, chamadas de **fade_in** e **fade_out**, assim como transições de movimento, chamadas de **slide_in** e **slide_out**.

Antes de continuar, remova todo o código de animação que fizemos no projeto dos planetas. Assim podemos partir de um código limpo para brincar com as novas animações do Android Lollipop.

Para começarmos, o código da `MainActivity` pode ficar simples assim:



MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void onClickPlaneta(View view) {
        Intent intent = new Intent(getBaseContext(), PlanetaActivity.class);
        startActivity(intent);
    }
}
```

E o código da `PlanetaActivity` pode ficar assim:



PlanetaActivity.java

```
public class PlanetaActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_planeta);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    }
}
```

Desta vez, para fazer a navegação de telas, vamos utilizar o método `makeSceneTransitionAnimation(activity)` da classe `ActivityOptions`, porém escolhemos a classe `ActivityOptionsCompat` para manter a compatibilidade.



MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // Habilita a transição de telas (deve ser a primeira linha de código)
        getWindow().requestFeature(Window.FEATURE_CONTENT_TRANSITIONS);
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

```

public void onClickPlaneta(View view) {
    Intent intent = new Intent(getBaseContext(), PlanetaActivity.class);
    // Efeito padrão de cross-fading
    ActivityOptionsCompat opts = ActivityOptionsCompat.makeSceneTransitionAnimation(this);
    ActivityCompat.startActivity(this, intent, opts.toBundle());
}
}

```

Dica: neste exemplo, estou habilitando a transição de activities dinamicamente no código. O método `requestFeature(FEATURE_CONTENT_TRANSITIONS)` deve ser chamado na primeira linha do método `onCreate(bundle)` de ambas as activities.

Na activity do planeta, também devemos habilitar a transição de telas na primeira linha do método `onCreate(bundle)`.



PlanetaActivity.java

```

public class PlanetaActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        getWindow().requestFeature(Window.FEATURE_CONTENT_TRANSITIONS);
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_planeta);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    }
}

```

Pronto! Execute o projeto e veja o efeito da animação. Por padrão, ao habilitar a animação de transição de telas, o efeito de sumir e aparecer (cross-fading) é criado automaticamente.

Agora vamos aprimorar o efeito de transição e animar a figura do planeta, conforme demonstrado na figura 11.15. O objetivo é redimensionar a foto do planeta Terra com uma harmoniosa animação durante a transição de activities, algo comum em aplicativos que seguem o Material Design.

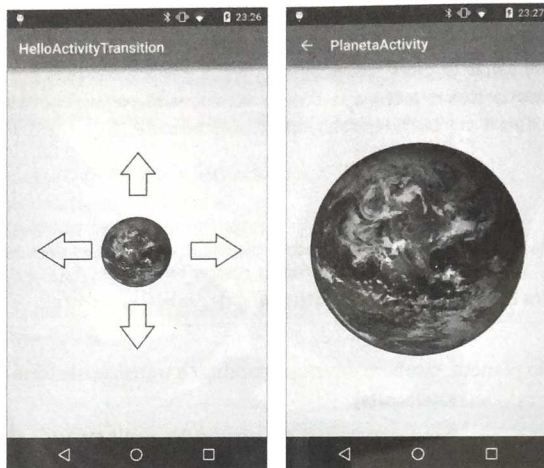


Figura 11.15 – Animação entre duas activities.

Para criar o efeito de transição, precisamos informar ao Android que existe um item compartilhado entre as activities. Isso é feito adicionando o atributo `android:transitionName` à view que você deseja compartilhar.

 `/res/layout/activity_main.xml`

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <ImageView android:id="@+id/img"
        android:transitionName="@string/transition_key"
        android:layout_gravity="center"
        android:src="@drawable/planeta_03_terra"
        android:layout_width="100dp" android:layout_height="100dp"
        android:onClick="onClickPlaneta" />
</FrameLayout>
```

A chave `@string/transition_key` pode ter qualquer texto e serve apenas para identificar esta view:

 `/res/values/strings.xml`

```
<resources>
    . . .
    <string name="transition_key">img_transition_key</string>
</resources>
```

Para o efeito funcionar, no layout da activity do planeta configure o `ImageView` de destino com a mesma chave de transição.

```

 /res/layout/activity_planeta.xml

<FrameLayout . . .>
    <ImageView android:id="@+id/img"
        android:transitionName="@string/transition_key" . . ./>
</FrameLayout>

```

Por último, para habilitar o compartilhamento das views durante a transição de telas, vamos passar a view e sua chave de compartilhamento como parâmetros para o método `makeSceneTransitionAnimation(activity,view,shareKey)`, conforme demonstrado a seguir.

MainActivity.java

```

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        // Habilita a transição de telas (deve ser a primeira linha de código)
        getWindow().requestFeature(Window.FEATURE_CONTENT_TRANSITIONS);
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void onClickPlaneta(View view) {
        Intent intent = new Intent(getBaseContext(), PlanetaActivity.class);
        // Compartilha a figura com o efeito de transição
        ImageView img = (ImageView) findViewById(R.id.img);
        String key = getString(R.string.transition_key);
        ActivityOptionsCompat opts = ActivityOptionsCompat.makeSceneTransitionAnimation(this,img,key);
        ActivityCompat.startActivity(this, intent, opts.toBundle());
    }
}

```

Pronto, isso é tudo. Agora execute o código novamente e veja a mágica acontecer.

Caso seja necessário, fazer a transição de mais uma view ao mesmo tempo também é possível, pois a assinatura do método `makeSceneTransitionAnimation(...)` pode receber uma lista de `android.support.v4.util.Pair<View, String>`.

```
makeSceneTransitionAnimation(Activity activity, Pair<View, String>... sharedElements)
```

Então digamos que no layout exista um `TextView` com o nome do planeta e a figura. Nesse caso, poderíamos fazer a transição de telas com este código:

```
Intent intent = new Intent(getBaseContext(), PlanetaActivity.class);
Pair p1 = Pair.create(findViewById(R.id.img), getString(R.string.img_transition_key));
Pair p2 = Pair.create(findViewById(R.id.img), getString(R.string.title_transition_key));
ActivityOptionsCompat opts = ActivityOptionsCompat.makeSceneTransitionAnimation(this, p1, p2);
ActivityCompat.startActivity(this, intent, opts.toBundle());
```

Se quiser conferir o resultado, abra o projeto **HelloActivityTransition-Pair** disponível nos exemplos do livro.

11.12 Compatibilidade com versões anteriores

Ao desenvolver aplicativos, uma importante decisão é qual será a versão mínima que você irá suportar. Durante o livro você vai aprender diversas técnicas para manter a compatibilidade, mas você pode perceber que na própria API já existem diversas classes como `ViewPage`, `ActionBar` de compatibilidade, `RecyclerView`, `CardView` etc. que já estão disponíveis para as versões mais antigas.

Uma dica é sempre procurar pela palavra "compat" quando você for precisar de alguma coisa, e provavelmente vai encontrar o que precisa.

Outro recurso muito utilizado é criar pastas com os qualificadores por API Level, como `/res/layout` e `/res/layout-21` (para API Level 21 ou superior).

E se você não puder customizar os arquivos XML utilizando esses qualificadores, é sempre possível executar um código customizado, fazendo o teste em tempo de execução.

```
// O Android 5.0 ou superior
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
    // Pode executar uma nova API
} else {
    // Implementa determinada funcionalidade de outra maneira.
}
```

11.13 Links úteis

Neste capítulo, estudamos o básico sobre o que é o Material Design. Recomendo que você olhe a documentação oficial para complementar seus estudos.

O site do Android Design é rico em conteúdo e contém muitas coisas sobre o Material Design, vale a pena conferir.

- **Google Design**
<http://www.google.com/design/>
- **Android Design**
<https://developer.android.com/design/>
- **Android Design – Material Design**
<http://www.google.com/design/spec/material-design/>
- **Android Design – Especificação do Material Design**
<https://developer.android.com/design/material/>
- **Blog do Google Developers – Post sobre Material Design**
<http://android-developers.blogspot.com.br/2014/10/implementing-material-design-in-your.html>
- **Android Training – Creating Apps with Material Design**
<https://developer.android.com/training/material/>
- **Android Training – Defining Custom Animations**
<https://developer.android.com/training/material/animations.html>
- **Android Developers Blog – Android Design Support Library**
<http://android-developers.blogspot.com.br/2015/05/android-design-support-library.html>



CAPÍTULO 12

Toolbar

Com o lançamento do Android 5.0 (Lollipop) foi criada a Toolbar, uma view especial que pode ser inserida em qualquer lugar do layout e tem as mesmas funcionalidades da action bar.

A Toolbar está sendo muito utilizada para criar aplicativos seguindo as boas práticas de interface do Material Design.

12.1 Introdução à Toolbar

Segundo a documentação do Android, a Toolbar é uma generalização da action bar, e sua vantagem é que ela é uma view que pode ser inserida em qualquer lugar do layout e tem as mesmas funcionalidades da action bar.

A action bar é um elemento fixo associado à activity e sempre fica no topo do layout. Já a Toolbar é uma view, sendo assim, ela pode ser inserida onde você desejar, inclusive ela pode aparecer mais de uma vez no layout. A figura 12.1 mostra um exemplo em que a Toolbar poderia ser utilizada em uma janela de dialog, pois ao abrir essa janela podemos ter a mesma barra de ações que temos na action bar.



Figura 12.1 – Conceito da Toolbar.

Outro exemplo de utilização da Toolbar pode ser visto na figura 12.2, que mostra um contato selecionado no aplicativo e a Toolbar mostra as ações que podemos fazer com aquele contato. Como podemos ver, a vantagem de utilizar a Toolbar é que ela é uma view parecida com a action bar, portanto os usuários vão reconhecer esse padrão e vão se sentir familiarizados com ele. A Toolbar pode mostrar as principais ações com os actions buttons e inclusive ter o menu flutuante action overflow com as opções menos utilizadas.



Figura 12.2 – Exemplo de Toolbar.

Como Toolbar é uma view, é comum os aplicativos criarem animações para mover a Toolbar ou redimensioná-la. Um exemplo disso é o aplicativo da agenda e ligação do Android 5.0. A figura 12.3 mostra um contato da agenda e ao selecioná-lo (clicar nos três pontinhos) uma view com a Toolbar aparece na tela com uma animação de baixo para cima (à direita da figura).

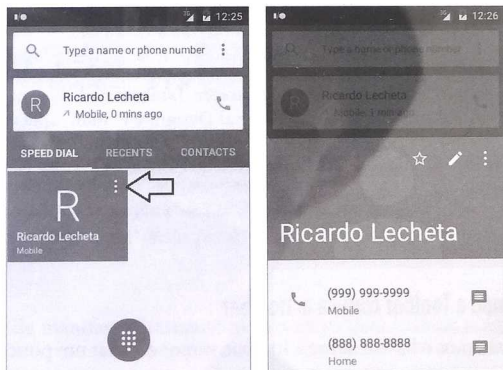


Figura 12.3 – Animação da Toolbar ao fazer a rolagem.

Nesse momento, se você tocar na view da Toolbar e arrastá-la para cima, todo o conteúdo aparece por cima da tela original, até a Toolbar se fixar no topo

(Figura 12.4). Caso o contato tenha muitas informações cadastradas e você continuar fazendo a rolagem para cima, a Toolbar é redimensionada com uma animação e fica pequena igual à action bar (à direita da figura).

Este exemplo mostrou o poder e a flexibilidade da Toolbar se comparada à action bar, e acredito que tenham ficado claras as diferenças. O recomendado é você brincar como esse aplicativo no emulador ou no seu smartphone para visualizar as animações de movimento que são muito fluidas.

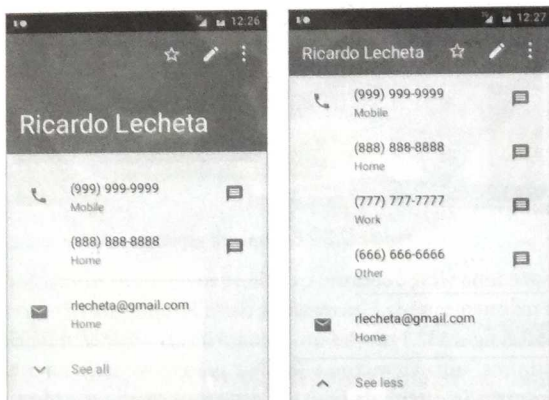


Figura 12.4 – Animação da Toolbar ao fazer a rolagem.

Nota: como a Toolbar é uma view, podemos executar animações de transparência com o objetivo de fazê-la aparecer e desaparecer. Também podemos movimentar a Toolbar conforme desejado. No Material Design, é comum aplicativos que utilizam listas monitorarem a rolagem da lista a fim de esconder a Toolbar para dar mais espaço ao conteúdo. Os aplicativos do Google I/O 2014 e Google Play utilizam esses conceitos.

12.2 Utilizando a Toolbar como a action bar

Agora que sabemos o básico sobre a Toolbar, vamos estudar um pouco de código.

A maneira mais simples de utilizar a Toolbar é desligar a action bar nativa, inserir a Toolbar no layout e configurá-la para tomar o lugar da action bar. Isso é simples, e consiste em três passos:

1. Para desligar a action bar, configure o aplicativo ou a activity desejada para utilizar o tema `Theme.AppCompat.NoActionBar` ou `Theme.AppCompat.Light.NoActionBar`.
2. Adicione a Toolbar em algum lugar do layout.
3. Utilize o método `setSupportActionBar(toolbar)` para transformar a Toolbar na action bar.

Para aprendermos isso na prática, crie um projeto chamado **HelloToolbar** no Android Studio e configure-o com o tema **AppCompat**. A classe `android.widget.Toolbar` foi criada a partir do Android 5.0, mas felizmente podemos utilizar a biblioteca de compatibilidade v7, que contém a classe `android.support.v7.widget.Toolbar`.

 **app/build.gradle**

```
dependencies {
    . . .
    compile 'com.android.support:appcompat-v7:22.1.0'
}
```

Neste próximo exemplo, vamos desabilitar a action bar da `MainActivity` e substituí-la pela Toolbar. Para começar, crie o tema `AppTheme.NoActionBar`.

 **/res/values/styles.xml**

```
<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <item name="colorPrimary">@color/primary</item>
        <item name="colorPrimaryDark">@color/primary_dark</item>
        <item name="colorAccent">@color/accent</item>
    </style>
    <!-- Desliga a action bar - NoActionBar -->
    <style name="AppTheme.NoActionBar" parent="Theme.AppCompat.Light.NoActionBar">
        <item name="colorPrimary">@color/primary</item>
        <item name="colorPrimaryDark">@color/primary_dark</item>
        <item name="colorAccent">@color/accent</item>
    </style>
</resources>
```

No arquivo de manifesto, configure a `MainActivity` para utilizar o tema `AppTheme.NoActionBar`.

 **AndroidManifest.xml**

```
<manifest . . . />
    <application android:theme="@style/AppTheme" >
```



```

<activity android:name=".MainActivity" android:theme="@style/AppTheme.NoActionBar">
    . . .
</activity>
</application>
</manifest>

```

Essa configuração vai remover a action bar da activity, portanto precisamos inserir a Toolbar no layout. Para isso, eu gosto de criar um arquivo de layout separado conforme demonstrado a seguir.



/res/layout/include_toolbar.xml

```

<android.support.v7.widget.Toolbar xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/toolbar"
    android:layout_height="wrap_content" android:layout_width="match_parent"
    android:minHeight="?attr/actionBarSize" android:background="?attr/colorPrimary" />

```

Nesse arquivo a altura da Toolbar está configurada como a altura da action bar. Isso é feito acessando o recurso nativo `?attr/actionBarSize`. A cor de fundo da Toolbar foi definida pelo recurso `?attr/colorPrimary`, ou seja, a cor primária do tema Material.

Nota: você deve ter reparado que a notação `?attr` é utilizada para acessar um recurso nativo do Android. Isso é interessante, pois podemos obter os valores padrões da plataforma.

Depois de criar o arquivo `/res/layout/include_toolbar.xml`, podemos incluí-lo no layout da activity. Isso é feito com a tag `<include>`.



/res/layout/activity_main.xml

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <!-- A Toolbar sempre fica no topo -->
    <include layout="@layout/include_toolbar" />
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/hello_world" />
</LinearLayout>

```

Por último, no código da activity é preciso chamar o método `setSupportActionBar(toolbar)` para transformar a Toolbar na action bar. Com isso o Android vai mostrar a Toolbar que inserimos no layout, porém todos os métodos da classe `ActionBar` continuam funcionando como antes. Podemos dizer que esse é um artifício técnico interessante.



MainActivity.java

```
public class MainActivity extends AppCompatActivity {  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
        // Aqui é a mágica (A Toolbar será a action bar).  
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);  
        setSupportActionBar(toolbar);  
    }  
    @Override  
    public boolean onCreateOptionsMenu(Menu menu) {  
        getMenuInflater().inflate(R.menu.menu_main, menu);  
        return true;  
    }  
    @Override  
    public boolean onOptionsItemSelected(MenuItem item) {  
        // Trate os eventos da action bar normalmente aqui.  
    }  
}
```

Pronto, isso é tudo. A figura 12.5 mostra o resultado. Parece que é uma action bar, mas na verdade é uma Toolbar. Inclusive o botão do smile funciona normalmente.

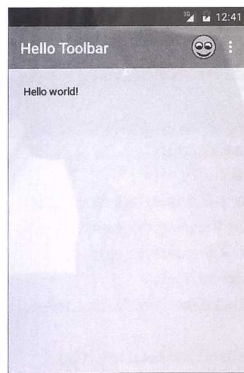


Figura 12.5 – Toolbar no lugar da action bar.

A vantagem de utilizar a Toolbar dessa forma no lugar da action bar é que a mesma API da classe ActionBar pode ser utilizada, então os métodos `onOptionsItemSelected()` e `onOptionsItemSelected()` continuam funcionando da mesma forma. Também podemos chamar o método `getSupportActionBar()` para recuperar o objeto da ActionBar e chamar qualquer método que quisermos.

Nota: utilizar a Toolbar como a action bar nos permite continuar utilizando os mesmos métodos com que já estamos acostumados da classe ActionBar. Com a vantagem de que podemos inserir a Toolbar em qualquer lugar do layout e ainda brincar de animá-la, pois ela é uma view como outra qualquer.

12.3 Utilizando a API da Toolbar (modo standalone)

Outra forma de utilizar a Toolbar é adicioná-la no layout e usar a sua própria API. Neste caso não importa se você desligou ou não a action bar nativa, pois inclusive ambas podem coexistir.

Para este próximo exemplo, crie o projeto **HelloToolbarStandalone** ou abra o projeto pronto disponível no material de download do livro. Para utilizar a Toolbar basta adicioná-la no layout como já fizemos. Mas desta vez, no código da activity, vamos usar a API da própria classe Toolbar.

MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        toolbar.setTitle(getString(R.string.app_name));
        toolbar.setNavigationIcon(R.drawable.ic_up);
        // Trata o evento de clique na Toolbar
        toolbar.setOnMenuItemClickListener(new Toolbar.OnMenuItemClickListener() {
            @Override
            public boolean onMenuItemClick(MenuItem item) {
                // Trate os eventos da action bar normalmente aqui.
            }
        });
    }
}
```

```

// Up Navigation
toolbar.setNavigationOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) { finish(); }
});
// Infla os itens de menu na Toolbar
toolbar.inflateMenu(R.menu.menu_main);
}
private Context getContext() { return this; }
}

```

No layout da activity vamos adicionar a Toolbar. Para brincar, ela foi inserida na parte inferior da tela.

 /res/layout/activity_main.xml

```

<FrameLayout . . .>
    <TextView . . ./>
    <!-- Toolbar na parte inferior -->
    <include layout="@layout/include_toolbar_light"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:layout_gravity="bottom" />
</FrameLayout>

```

O resultado pode ser visto na figura 12.6. Veja que a activity mostrou a action bar normalmente lá em cima. Na parte inferior podemos ver a Toolbar, que inflou os itens de menu e também ativou o up navigation.

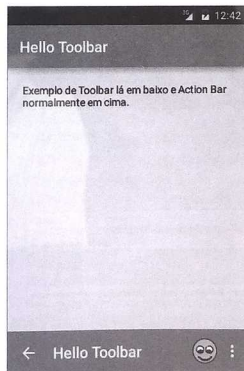


Figura 12.6 – Toolbar inserida na parte inferior do layout.

12.4 Links úteis

Neste capítulo, estudamos o básico sobre a classe `Toolbar` e para continuar seus estudos selecionei alguns links da documentação oficial.

- **Toolbar – Documentação da API**

<https://developer.android.com/reference/android/support/v7/widget/Toolbar.html>

- **Android Design – Toolbar**

<http://www.google.com/design/spec/components/toolbars.html>

- **Blog do Google Developers – Post sobre a Toolbar**

<http://android-developers.blogspot.com.br/2014/10/appcompat-v21-material-design-for-pre.html>

- **Blog do Google Developers – Post sobre Material Design**

<http://android-developers.blogspot.com.br/2014/10/implementing-material-design-in-your.html>



CAPÍTULO 13

Navigation Drawer

Neste capítulo, começaremos o desenvolvimento do projeto dos carros, no qual vamos praticar os conceitos aprendidos até o momento e aprender muito mais. Na sequência dos próximos capítulos, o aplicativo será incrementado com novas funcionalidades.

Neste momento, vamos criar o projeto e preparar a estrutura básica para utilizar o Navigation Drawer (menu lateral deslizante) como o padrão de navegação.

13.1 Criando o projeto

Para começar a brincadeira, crie o projeto conforme a figura 13.1. Preenchi o campo **Company Domain** com **livroandroid.com.br** e recomendo que você faça o mesmo, pois assim o pacote gerado no seu projeto ficará com o mesmo nome dos meus códigos, facilitando qualquer **copy-paste** que você venha a fazer ao seguir os exemplos.

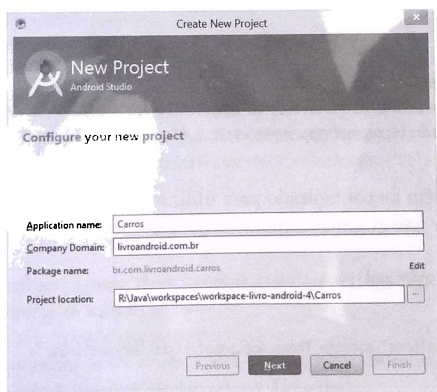


Figura 13.1 – Criando o projeto dos carros.

No wizard selecione o Android 2.3 como a versão mínima (Figura 13.2). Clique em **Next** e na próxima página do wizard escreva **Carros** no campo **Title**.

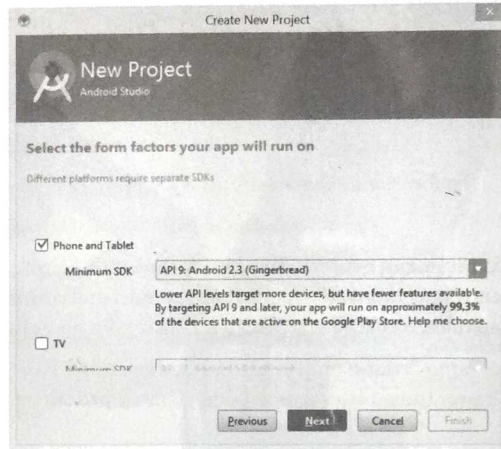


Figura 13.2 – Criando o projeto dos carros.

Depois de criar o projeto, como selecionamos o Android 2.3 (API Level 9) para a versão mínima suportada pelo aplicativo, o Android Studio já configurou o projeto para utilizar a biblioteca de compatibilidade v7.

 **app/build.gradle**

```
...
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    // Ativa a dependência da biblioteca de compatibilidade v7
    compile 'com.android.support:appcompat-v7:22.1.1'
}
```

O projeto também foi configurado para utilizar o tema AppCompat. Se por acaso o wizard não fizer isso automaticamente, faça estas alterações:

 **/res/values/styles.xml**

```
<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    </style>
</resources>
```

Dica: para facilitar a digitação do código-fonte no Android Studio e a inserção dos imports no código, vamos ativar os imports automáticos. Abra o menu File > Settings, entre no menu Editor > Auto Import e deixe ligada a opção Add unambiguous imports on the fly.

13.2 Customizando as cores do tema Material

Conforme já estudamos anteriormente, podemos customizar as cores do tema Material facilmente, portanto crie o arquivo com as cores.

 /res/values/colors.xml

```
<resources>
    <!-- Cores do Tema Material -->
    <color name="primary">#03A9F4</color>
    <color name="primary_dark">#01579B</color>
    <color name="accent">#F44336</color>
    <color name="control_highlight">#B3E5FC</color>
    <!-- Outras cores -->
    <color name="transparent">#0000</color>
    <color name="white">#fff</color>
    <color name="black">#000</color>
    <color name="blue">#00f</color>
    <color name="red">#f00</color>
    <color name="green">#0f0</color>
    <color name="gray">#eee</color>
</resources>
```

No arquivo de cores já aproveitei e criei algumas cores comuns que podemos utilizar no aplicativo. Para prosseguir, configure o tema com as cores do tema Material, que são os atributos colorPrimary, colorPrimaryDark, colorAccent e colorControlHighlight.

 /res/values/styles.xml

```
<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
        <!-- Cor primária do aplicativo -->
        <item name="colorPrimary">@color/primary</item>
        <!-- Variação escura da cor primária para a status bar e contextual app bars -->
        <item name="colorPrimaryDark">@color/primary_dark</item>
```



```

<!-- Cor de acentuação para as views (checkbox, textfield etc.) -->
<item name="colorAccent">@color/accent</item>
<!-- Cor para o efeito ripple -->
<item name="colorControlHighlight">@color/control_highlight</item>
</style>
</resources>

```

Antes de executar, garanta que no arquivo `/res/values/strings.xml` a chave `app_name` esteja com o texto **Carros**, assim o título da action bar vai ficar legal.

 `/res/values/strings.xml`

```

<resources>
    <string name="app_name">Carros</string>
    <string name="hello_world">Hello world!</string>
    <string name="action_settings">Settings</string>
</resources>

```

A figura 13.3 mostra o projeto executando no emulador. Este livro não foi impresso com cores, portanto certifique-se de que a action bar ficou com as cores que você definiu no tema.

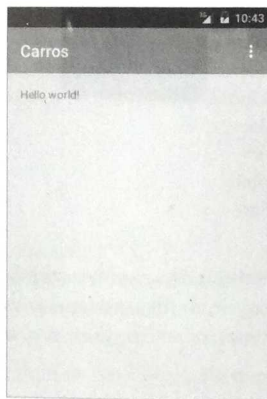


Figura 13.3 – Cores da action bar customizadas.

13.3 Criando a activity e o fragment base para o projeto

Algo que eu sempre gosto de fazer nos meus projetos é criar uma classe de activity mãe para todas as activities do projeto, e o mesmo vale para os fragments.

Então vamos lá. Crie a classe `BaseActivity`, que é filha de `AppCompatActivity`. Utilize o menu **New > Java Class**. Essa classe deve ficar no pacote `br.livroandroid.carros.activity`.

BaseActivity.java

```
package br.com.livroandroid.carros.activity;
import android.support.v7.app.AppCompatActivity;
public class BaseActivity extends AppCompatActivity {
}
```

A vantagem de fazer isso é podermos colocar métodos na classe `BaseActivity` para reutilizarmos em todas as suas subclasses. Como até o momento somente a `MainActivity` existe no projeto, altere o seu código para herdar de `BaseActivity`.

MainActivity.java

```
package br.com.livroandroid.carros.activity;
public class MainActivity extends BaseActivity {
}
```

Ainda não temos nenhum fragment no projeto, mas já deixe criada a classe `BaseFragment`, conforme demonstrado a seguir. Essa classe deve ficar no pacote `br.livroandroid.carros.fragments`.

BaseFragment.java

```
package br.com.livroandroid.carros.fragments;
import android.support.v4.app.Fragment;
public class BaseFragment extends Fragment {
}
```

Para efeitos de organização do projeto, eu gosto de utilizar cada classe em seu respectivo pacote (Figura 13.4). Veja que deixo as classes de activity no pacote `br.livroandroid.carros.activity` e os fragments em `br.livroandroid.carros.fragments`.

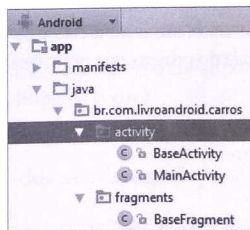


Figura 13.4 – Organização dos pacotes.

Lembrando que no código o pacote é a primeira linha do arquivo. Se você digitar um pacote que não existe no projeto, o Android Studio vai perguntar se você não deseja mover o arquivo, portanto ele vai ajudá-lo.

Dica: caso o editor mostre uma linha vermelha é porque o Android Studio identificou um problema. Nesses casos, é recomendado abrir o assistente com o atalho **Ctrl+Enter** para visualizar as opções. Um exemplo disso é quando digitamos no código um pacote que não existe (ex.: `br.com.livroandroid.carros.fragments`) e o assistente pode ajudar a mover a classe para o pacote correto.

Não se esqueça: ao fazer essa configuração de pacotes, no arquivo *AndroidManifest.xml*, você precisa configurar o caminho da *activity* relativa ao pacote principal, conforme demonstrado a seguir.

```
<activity android:name=".activity.MainActivity" ... />
```

Antes de prosseguir com a leitura, execute o projeto novamente e garanta que está tudo funcionando.

13.4 Classe Application – armazenando informações globais

É comum desenvolvedores utilizarem variáveis de classe (estáticas) para armazenar informações globais da aplicação, mas no Android não é recomendado fazer isso. O correto é criar uma classe de *Application* customizada que herda de *android.app.Application*, a qual faz parte do ciclo de vida da aplicação, e o Android vai criar essa classe junto com o processo da aplicação.

A classe *Application* é um **Singleton** que pode ser utilizado para armazenar informações de forma global no aplicativo. Caso você não saiba o que é um **Singleton**, segue uma breve explicação:

<http://pt.wikipedia.org/wiki/Singleton>

O código-fonte a seguir mostra como criar a classe *CarrosApplication*. Eu gosto de deixá-la sempre no pacote raiz do projeto, que neste caso é `br.com.livroandroid.carros`.

 **CarrosApplication.java**

```
package br.com.livroandroid.carros;
import android.app.Application;
import android.util.Log;
public class CarrosApplication extends Application {
    private static final String TAG = "CarrosApplication";
```

```

private static CarrosApplication instance = null;
public static CarrosApplication getInstance() {
    return instance; // Singleton
}
@Override
public void onCreate() {
    Log.d(TAG, "CarrosApplication.onCreate()");
    // Salva a instância para termos acesso como Singleton
    instance = this;
}
@Override
public void onTerminate() {
    super.onTerminate();
    Log.d(TAG, "CarrosApplication.onTerminate()");
}
}

```

Essa classe precisa ser registrada no *AndroidManifest.xml*, portanto configure a tag `<application>` conforme demonstrado a seguir:

AndroidManifest.xml

```

<manifest . . .
    <application android:name=".CarrosApplication" . . .
</manifest>

```

Dica: sempre depois de colocar qualquer nome de classe em algum arquivo XML, pressione Ctrl+Clique para abrir o arquivo. Se o arquivo abrir é porque a configuração está correta. Isso pode identificar erros simples de digitação.

Ao fazer essa configuração, a classe `CarrosApplication` será instanciada quando o processo da aplicação for criado. Nesse momento, o método `onCreate()` é chamado e estamos retendo a instância desse objeto. Quando o Android terminar o processo da aplicação, o método `onTerminate()` será chamado para limpar os recursos.

Assim, sempre que precisarmos acessar essa classe para salvar ou ler informações globais, basta utilizar esta linha de código:

```
CarrosApplication app = CarrosApplication.getInstance();
```

Mais para frente, durante o desenvolvimento do projeto dos carros, vamos utilizar essa classe.

13.5 Biblioteca android-utils

No projeto dos carros vamos utilizar a biblioteca `android-utils` que contém algumas classes utilitárias criadas para facilitar os exemplos e também o seu aprendizado.

O código-fonte da biblioteca `android-utils` está no GitHub para sua consulta:

<https://github.com/livroandroid/AndroidUtils/>

O objetivo dessa biblioteca é facilitar seu aprendizado e principalmente deixar o desenvolvimento do aplicativo dos carros mais fácil. Para utilizar a biblioteca `android-utils`, basta adicionar a dependência no arquivo `app/build.gradle`.

`app/build.gradle`

```
apply plugin: 'com.android.application'

...
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:22.1.0'
    compile 'br.com.livroandroid:android-utils:1.0.0'
}
```

Logo depois de adicionar essa linha no arquivo `app/build.gradle`, clique no botão **Sync Now** que aparece no editor. Isso fará com que o Gradle baixe a dependência e deixe-a disponível no projeto. Para testar se a dependência da biblioteca `android-utils` foi corretamente configurada, altere a classe `BaseActivity` do projeto dos carros para herdar de `livroandroid.lib.activity.BaseActivity`.

`BaseActivity.java`

```
package br.com.livroandroid.carros.activity;

public class BaseActivity extends livroandroid.lib.activity.BaseActivity {
}
```

Repare que as classes têm o mesmo nome, mas estão em pacotes diferentes. Se você tiver dúvidas sobre os pacotes, procure alguma leitura complementar. Da mesma forma, altere a classe `BaseFragment` que criamos para ser filha de `livroandroid.lib.fragment.BaseFragment` da biblioteca `android-utils`.

`BaseFragment.java`

```
package br.com.livroandroid.carros.fragments;

public class BaseFragment extends livroandroid.lib.fragment.BaseFragment {
}
```

Feito isso, salve os arquivos e compile o código com a opção **Build > Rebuild Project**. Se o projeto compilar, tudo está configurado corretamente, e as classes da biblioteca `android-utils` foram encontradas.

Caso você tenha dificuldades em seguir os passos descritos até aqui, recomendando comparar com o projeto **Passo01-CriarProjeto** disponível no site <https://github.com/livroandroid/carros>. Sempre que necessário, confira o seu código com os projetos de exemplo, os quais apresentam a solução para os exercícios que vamos fazer.

Nota: a classe `BaseActivity` da biblioteca `android-utils` contém alguns métodos auxiliares para mostrar alertas e toasts para facilitar a digitação dos exemplos. A classe `BaseFragment` da biblioteca contém os mesmos métodos, mas também tem o método `startTask(...)` que vamos utilizar posteriormente ao trabalhar com web services e banco de dados. Lembrando que criei essa biblioteca para facilitar os exemplos, e o código-fonte está disponível no site <https://github.com/livroandroid/AndroidUtils>.

13.6 Como o Gradle encontrou a biblioteca `android-utils`

Você deve estar se perguntando como o Gradle fez para encontrar a biblioteca `android-utils` que declaramos no arquivo `app/build.gradle`. Isso foi possível porque a biblioteca `android-utils` está instalada (foi feito o `deploy`) no Maven Central, que é um repositório mundial de projetos utilizado pelo Gradle para buscar as dependências.

Para validar que a biblioteca existe no repositório do Maven Central, acesse o endereço <http://search.maven.org> e faça a busca por `br.com.livroandroid` (Figura 13.5). Você também pode adicionar repositórios locais na sua máquina, ou até instalar um servidor de repositórios privado na rede de sua empresa.

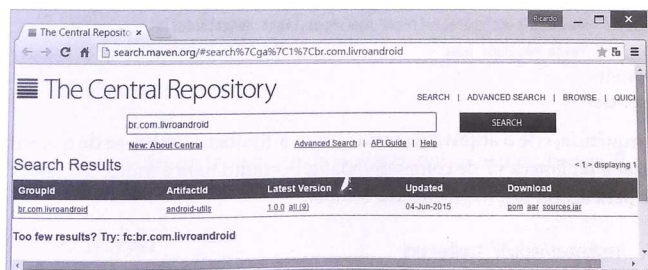


Figura 13.5 – Biblioteca `android-utils` no Maven Central.

Como fazer o deploy no Maven Central está fora do escopo do livro, mas no capítulo 38, sobre Gradle, vamos aprender a criar e utilizar projetos do tipo biblioteca. Também publiquei no meu site (ricardolecheta.com.br) uma série de artigos sobre como criar bibliotecas no Gradle, desde deixar uma biblioteca de forma local no seu computador, até fazer o deploy no Maven Central. Leia os artigos quando achar necessário, pois um dia você vai precisar criar suas próprias bibliotecas.

Nota: sempre que adicionar uma nova dependência no arquivo *app/build.gradle*, clique no botão **Sync Now** no editor. Isso faz com que o Gradle faça o download da dependência e a deixe ativa no projeto, para que as funcionalidades como o assistente de código e o compilador encontrem as classes dessa biblioteca. Essa opção também pode ser acessada no menu **Tools > Android > Sync Project with Gradle Files**.

13.7 Configurando a Toolbar

No aplicativo dos carros, vamos utilizar a Toolbar para termos maior controle sobre como ela é posicionada e até podemos fazer algumas animações de que vamos precisar mais para frente.

Conforme já estudamos no capítulo 12, sobre Toolbar, existem duas maneiras de utilizá-la. Vou optar pela mais simples, que é apenas desligar a action bar no tema e fazer a Toolbar ser reconhecida pelo sistema como a action bar.

Para isso, altere o tema do projeto para utilizar a versão `NoActionBar`.

 `/res/values/styles.xml`

```
<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
        // O resto não muda aqui
    </style>
</resources>
```

Na sequência, crie o arquivo de include com a Toolbar. Lembre-se de que vamos utilizar a biblioteca v7 de compatibilidade, portanto nunca utilize as classes nativas para action bar, fragments ou Toolbar.

 `/res/layout/include_toolbar.xml`

```
<android.support.v7.widget.Toolbar xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/toolbar">
```

```

    android:layout_width="match_parent" android:layout_height="wrap_content"
    android:background="?attr/colorPrimary" android:minHeight="?attr/actionBarSize"
    app:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
    app:popupTheme="@style/ThemeOverlay.AppCompat.Light" />

```

Feito isso, deixe o arquivo de layout da activity conforme demonstrado a seguir. Veja que o alterei para utilizar um `LinearLayout` com orientação vertical e tirei todos os espaçamentos (padding).

 /res/layout/activity_main.xml

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical">
    <!-- Toolbar -->
    <include layout="@layout/include_toolbar" />
    <TextView android:text="@string/hello_world"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
</LinearLayout>

```

Para finalizar a configuração, precisamos ativar a `Toolbar` no código. Como essa tarefa precisa ser feita em todas as activities, vamos criar o método `setUpToolbar()` na classe `BaseActivity`, que é mãe de todas as activities do projeto.

 BaseActivity.java

```

package br.com.livroandroid.carros.activity;

import android.support.v7.widget.Toolbar;

public class BaseActivity extends livroandroid.lib.activity.BaseActivity {
    protected void setUpToolbar() {
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        if (toolbar != null) {
            setSupportActionBar(toolbar);
        }
    }
}

```

Em todas as activities do projeto, não podemos nos esquecer de ativar a `Toolbar`, portanto adicione a seguinte linha de código na classe `MainActivity`:

 MainActivity.java

```

package br.com.livroandroid.carros.activity;

public class MainActivity extends BaseActivity {
    @Override

```



```
protected void onCreate(Bundle savedInstanceState) {  
    super.onCreate(savedInstanceState);  
    setContentView(R.layout.activity_main);  
    setUpToolbar();  
}  
}
```

Pronto. Feito isso, execute o projeto, e tudo deve continuar funcionando. A diferença é que estamos usando a Toolbar no lugar da action bar. A vantagem é que a Toolbar é uma view normal, portanto podemos criar animações de movimento ou transparência quando for necessário. Para ter ideia dessas animações, brinque um pouco com o aplicativo do Google Play e veja os efeitos que ele faz com a Toolbar.

13.8 Navigation Drawer

Definir o padrão de navegação do aplicativo é importante para facilitar a navegação das telas. Para auxiliar nessa tarefa, o Android apresenta alguns padrões bem conhecidos pelos usuários, como a navegação por tabs e o Navigation Drawer (menu lateral deslizante).

A navegação por tabs já estudamos, e ela é recomendada quando existem poucas ações, pois o usuário pode ver as tabs disponíveis e descobrir rapidamente o que o aplicativo pode fazer. Caso o aplicativo tenha muitas ações, é recomendado utilizar o Navigation Drawer, pois o menu lateral pode abrir uma listagem de vários itens, facilitando a visualização do usuário.

No caso do aplicativo dos carros, não temos muitas ações, mas vamos utilizar o padrão Navigation Drawer mesmo assim, para praticarmos. Internamente no aplicativo para navegar entre os carros esportivos, clássicos e luxo, vamos utilizar as tabs. O padrão Navigation Drawer é utilizado em vários aplicativos nativos do Android, como Gmail, Google Play, Maps, Google Drive, YouTube, Play Music, Photos, entre outros, e recomendo que você leia a documentação oficial para aprender mais detalhes.

<http://www.google.com/design/spec/patterns/navigation-drawer.html>

A figura 13.6 mostra o que queremos fazer neste capítulo.

Criar um Navigation Drawer não é tão simples, principalmente se formos seguir as boas práticas do Material Design. Segundo essas boas práticas, o Navigation Drawer precisa ser aberto por cima da action bar, ocupando todo o espaço disponível no celular. O menu tem diversas métricas de espaçamento e alinhamento que você precisa seguir, portanto recomendo que você leia a página do Android Design para obter mais informações.

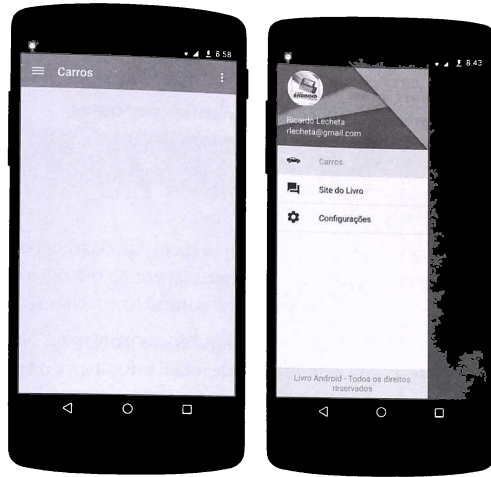


Figura 13.6 – Navigation Drawer no projeto dos carros.

Antigamente, o Navigation Drawer abria por baixo da action bar, algo que até que era simples de fazer. No Material Design, como temos de abrir o menu lateral por cima de tudo, vamos precisar fazer algumas mágicas no código. Mas fique tranquilo que vou ajudar você a fazer tudo passo a passo.

Então, mãos à obra!

Altere o arquivo de layout da activity principal, conforme demonstrado a seguir:

```

<?xml version="1.0" encoding="utf-8" ?>
<android.support.v4.widget.DrawerLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent" android:layout_height="match_parent" >
    <!-- Bloco 1: Conteúdo da Tela -->
    <LinearLayout android:id="@+id/content"
        android:layout_width="match_parent" android:layout_height="match_parent"
        android:orientation="vertical">
        <include layout="@layout/include_toolbar" />
    <!-- Conteúdo -->
    <FrameLayout
        android:id="@+id/nav_drawer_container"
        android:layout_width="match_parent" android:layout_height="match_parent" />

```

```

</LinearLayout>
<!-- Bloco 2: Nav Drawer (menu lateral deslizante) -->
<fragment
    android:id="@+id/nav_drawer_fragment"
    android:name="livroandroid.lib.fragment.NavigationDrawerFragment"
    android:layout_width="@dimen/navigation_drawer_width"
    android:layout_height="match_parent"
    android:layout_gravity="start" />
</android.support.v4.widget.DrawerLayout>

```

A classe `DrawerLayout` é responsável por abrir o menu deslizante, por isso ela foi definida como a raiz deste layout. Essa classe contém os métodos `openDrawer()` e `closeDrawer()`, que podemos utilizar no código para abrir e fechar o menu.

No layout do arquivo XML, são definidos dois blocos principais. Na parte superior é definido o conteúdo do aplicativo, onde inclui a `Toolbar`, e na parte inferior existe um `FrameLayout` que ficará vazio por enquanto, pois será utilizado para incluir os `fragments` ao selecionar algum item do menu.

Na parte inferior do layout, foi adicionado o `fragment NavigationDrawerFragment`, que faz parte da biblioteca `android-utils`. Vamos utilizar essa classe da biblioteca para facilitar a criação do `Navigation Drawer`. Uma vez que alteramos o arquivo de layout, vamos atualizar o código-fonte da `activity`, conforme demonstrado a seguir.



MainActivity.java

```

public class MainActivity extends BaseActivity {
    private NavigationDrawerFragment mNavDrawerFragment;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        setUpToolbar();
        // Nav Drawer
        mNavDrawerFragment = (NavigationDrawerFragment) getSupportFragmentManager()
            .findFragmentById(R.id.nav_drawer_fragment);
        // Configura o drawer layout
        DrawerLayout drawerLayout = (DrawerLayout) findViewById(R.id.drawer_layout);
        mNavDrawerFragment.setUp(drawerLayout);
    }
}

```

Esse código simplesmente obtém os objetos `NavigationDrawerFragment` e `DrawerLayout` que estão no layout. O método `setUp(drawerLayout)` é utilizado para configurar o `NavigationDrawerFragment` com o `DrawerLayout` que foi adicionado no arquivo de layout.

Feito isso, execute o código, e o resultado deve ser como a figura 13.7. A parte esquerda da figura mostra o menu fechado e na parte da direita podemos ver o menu aberto. O menu lateral tem o fundo transparente, por isso precisamos definir o conteúdo agora para o visual ficar mais interessante.

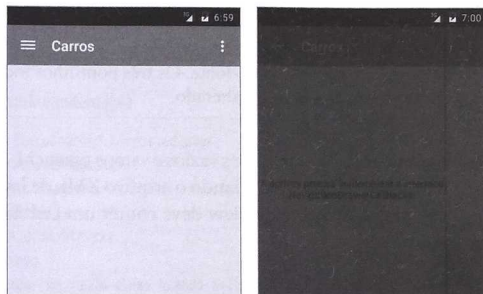


Figura 13.7 – Navigation Drawer.

Para criar a lista no menu lateral, vamos fazer a activity implementar a interface `NavigationDrawerFragment.NavigationDrawerCallbacks`. Essa interface é utilizada na biblioteca `android-utils` para que a activity possa retornar a view do menu lateral. Portanto, altere o código-fonte da classe `MainActivity` conforme demonstrado a seguir. Veja que coloquei comentários para explicar o significado de cada método.

MainActivity.java

```
public class MainActivity extends BaseActivity implements
    NavigationDrawerFragment.NavigationDrawerCallbacks {
    ...
    @Override
    public NavigationDrawerFragment.NavDrawerListView
        getNavDrawerView(NavigationDrawerFragment navigationDrawerFragment,
            LayoutInflater inflater, ViewGroup container) {
        // Deve retornar a view e o identificador do ListView
        return null;
    }
    @Override
    public ListAdapter getNavDrawerListAdapter(NavigationDrawerFragment navigationDrawerFragment) {
        // Este método deve retornar o adapter que vai preencher o ListView
        return null;
    }
    @Override
```

```

public void onNavDrawerItemSelected(NavigationDrawerFragment navigationDrawerFragment,
    int position) {
    // Método chamado ao selecionar um item do ListView.
}
}

```

Dica: nos códigos do livro, utilizo a notação de três pontinhos (. . .). O objetivo é mostrar apenas a parte nova do código-fonte. Os três pontinhos indicam que o restante do código-fonte não deve ser alterado.

Por enquanto, deixaremos estes três métodos vazios e vamos preenchê-los à medida que criarmos os arquivos. Começamos criando o arquivo XML de layout da view que vai aparecer no menu lateral. Essa view deve conter um `ListView`, conforme demonstrado a seguir:

 `/res/layout/nav_drawer_listview.xml`

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/listViewContainer" android:orientation="vertical"
    android:layout_width="match_parent" android:layout_height="match_parent" >
    <ListView android:id="@+id/listView"
        android:layout_width="match_parent" android:layout_height="0dp"
        android:layout_weight="1" android:background="#ffffff" />
    <TextView
        android:layout_width="match_parent" android:layout_height="50dp"
        android:background="@color/gray" android:gravity="center"
        android:text="Livro Android - Todos os direitos reservados"
        android:textColor="?attr/colorPrimary" />
</LinearLayout>

```

Veja que, no código deste arquivo de layout, criei o `ListView` e um `TextView` com um texto sobre direitos reservados. Com esse arquivo pronto, podemos implementar o método `getNavDrawerView(...)` da interface `NavigationDrawerCallbacks`, o qual deve inflar o layout para criar a view, e retornar a view e o identificador do `ListView`. Dessa forma, a biblioteca `android-utils` vai saber como encontrar o `ListView` no layout.

 `MainActivity.java`

```

public class MainActivity . . . {
    . . .
    @Override
    public NavigationDrawerFragment.NavDrawerListView getNavDrawerView(..) {
        View view = LayoutInflater.inflate(R.layout.nav_drawer_listview, container, false);
    }
}

```

```

        return new NavigationDrawerFragment.NavDrawerListView(view,R.id.listView);
    }
}

```

Se você executar o projeto agora e abrir o menu lateral, verá que esse layout já foi utilizado, porém o `ListView` está vazio. Para preencher o `ListView`, vamos precisar de um objeto que contenha o título e a figura de cada item do menu, portanto crie a classe `NavDrawerMenuItem` no pacote `adapter`.

NavDrawerMenuItem.java

```

package br.com.livroandroid.carros.adapter;

public class NavDrawerMenuItem {
    // Título: R.string.xxx
    public int title;
    // Figura: R.drawable.xxx
    public int img;
    // Para colocar um fundo cinza quando a linha está selecionada
    public boolean selected;
    public NavDrawerMenuItem(int title, int img) {
        this.title = title;
        this.img = img;
    }
    // Cria a lista com os itens de menu
    public static List<NavDrawerMenuItem> getList() {
        List<NavDrawerMenuItem> list = new ArrayList<NavDrawerMenuItem>();
        list.add(new NavDrawerMenuItem(R.string.carros, R.drawable.ic_drawer_carro));
        list.add(new NavDrawerMenuItem(R.string.site_livro, R.drawable.ic_drawer_site_livro));
        list.add(new NavDrawerMenuItem(R.string.configuracoes, R.drawable.ic_drawer_settings));
        return list;
    }
}

```

Para esse código funcionar, não se esqueça de copiar as imagens dos ícones do menu para o projeto. Você pode encontrar todas as imagens de que vamos precisar nos projetos de exemplo de cada capítulo. Adicione também os títulos de cada item de menu no arquivo `/res/values/strings.xml`.

/res/values/strings.xml

```

<resources>
    .
    .
    .
    <string name="carros">Carros</string>
    <string name="site_livro">Site do Livro</string>
    <string name="configuracoes">Configurações</string>
</resources>

```

O arquivo de layout do adapter pode ser visualizado a seguir. Ele contém apenas o `ImageView` do ícone e o `TextView` para o título do item de menu.

 `/res/layout/adapter_nav_drawer.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="48dp"
    android:background="@color/white" android:gravity="center_vertical"
    android:orientation="horizontal"
    android:paddingLeft="16dp" android:paddingRight="16dp">
    <ImageView android:id="@+id/img"
        android:layout_width="24dp" android:layout_height="24dp"
        android:layout_marginRight="32dp"
        android:src="@drawable/ic_drawer_carro" />
    <TextView android:id="@+id/text"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="Menu"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:textColor="@color/black" android:textSize="14sp" />
</LinearLayout>
```

A seguir, temos o código da classe do adapter, que vai preencher o `ListView`. Você pode criá-la no pacote `adapter`.

 `NavDrawerMenuAdapter.java`

```
package br.com.livroandroid.carros.adapter;
// imports
public class NavDrawerMenuAdapter extends BaseAdapter {
    protected static final String TAG = "livroandroid";
    private final List<NavDrawerMenuItem> list;
    private final Context context;
    private LayoutInflater inflater;
    public NavDrawerMenuAdapter(Context context, List<NavDrawerMenuItem> list) {
        this.context = context;
        this.list = list;
        this.inflater = (LayoutInflater) LayoutInflater.from(context);
    }
    @Override
    public int getCount() {
        return list != null ? list.size() : 0;
    }
    @Override
```

```

    public Object getItem(int position) {
        return list != null ? list.get(position) : null;
    }
    @Override
    public long getItemId(int position) {
        return position;
    }
    @Override
    public View getView(int position, View view, ViewGroup parent) {
        ViewHolder holder = null;
        if (view == null) {
            // Cria o ViewHolder
            holder = new ViewHolder();
            view = inflater.inflate(R.layout.adapter_nav_drawer, parent, false);
            view.setTag(holder);
            holder.text = (TextView) view.findViewById(R.id.text);
            holder.img = (ImageView) view.findViewById(R.id.img);
        } else {
            // Reaproveita o ViewHolder
            holder = (ViewHolder) view.getTag();
        }
        // Atualiza a view
        NavDrawerMenuItem item = list.get(position);
        holder.text.setText(item.title);
        holder.img.setImageResource(item.img);
        if (item.selected) {
            // Configura o fundo cinza do item selecionado.
            view.setBackgroundResource(R.drawable.seletor_nav_drawer_selected);
            holder.text.setTextColor(context.getResources().getColor(R.color.primary));
        } else {
            view.setBackgroundResource(R.drawable.seletor_nav_drawer);
            holder.text.setTextColor(context.getResources().getColor(R.color.black));
        }
        return view;
    }
    public void setSelected(int position, boolean selected) {
        clearSelected();
        list.get(position).selected = selected;
        notifyDataSetChanged();
    }
    public void clearSelected() {
        if (list != null) {
            for (NavDrawerMenuItem item : list) {

```



```

        item.selected = false;
    }
    notifyDataSetChanged();
}
}
// Design Patter "ViewHolder"
static class ViewHolder {
    TextView text;
    ImageView img;
}
}

```

O código da classe do adapter é extenso, mas acredito que você já consiga entendê-lo. O adapter não apenas está inflando seu arquivo de layout para criar a view, como também utiliza o padrão `ViewHolder` para reaproveitar as views durante a rolagem. Observe que também coloquei os métodos `setSelected()` e `clearSelected()` para controlar o item de menu que está selecionado, pois é uma boa prática mostrar um fundo com alguma cor diferenciada quando algum item está selecionado. Justamente por causa desse fundo, precisamos criar os arquivos de seletores conforme demonstrado a seguir:

 /res/drawable/seletor_nav_drawer.xml

```

<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@color/transparent" android:state_selected="true" />
    <item android:drawable="@color/transparent" android:state_pressed="true" />
    <item android:drawable="@color/white" android:state_selected="false" />
</selector>

```

 /res/drawable/seletor_nav_drawer_selected.xml

```

<selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:drawable="@color/transparent" android:state_selected="true" />
    <item android:drawable="@color/transparent" android:state_pressed="true" />
    <item android:drawable="@color/gray" android:state_selected="false" />
</selector>

```

Nota: um seletor (selector) é um arquivo XML que deve ficar na pasta `/res/drawable`. Para cada estado, como selecionado (selected) ou pressionado (pressed), podemos definir o fundo necessário, seja com imagens ou cores. Neste caso, estou definindo a cor de fundo cinza quando o item está selecionado.

Depois de criar o adapter, implemente o método `getNavDrawerListAdapter(navFrag)` da interface `NavigationDrawerCallbacks`.

```

MainActivity.java

public class MainActivity extends BaseActivity implements
    NavigationDrawerFragment.NavigationDrawerCallbacks {
    private NavDrawerMenuAdapter listAdapter;
    ...
    @Override
    public ListAdapter getNavDrawerListAdapter(NavigationDrawerFragment
        navigationDrawerFragment) {
        List<NavDrawerMenuItem> list = NavDrawerMenuItem.getList();
        // Deixa o primeiro item selecionado
        list.get(0).selected = true;
        this.listAdapter = new NavDrawerMenuAdapter(this, list);
        return listAdapter;
    }
}

```

Terminada essa alteração, execute o projeto novamente. Dessa vez, o menu lateral vai mostrar a lista com as opções, conforme a figura 13.8.

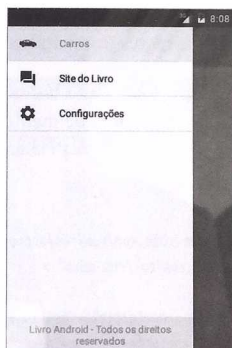


Figura 13.8 – Navigation Drawer com os itens de menu.

Nota: se tivéssemos utilizado a action bar, o menu lateral seria aberto por debaixo da barra. Como utilizamos a Toolbar e ela é uma view, o menu lateral consegue abrir por cima da Toolbar, o que é um padrão do Material Design. Mesmo assim, ele não abriu por cima da status bar (barra superior que mostra as horas), mas isso vamos ver depois.

Para finalizar a criação do Navigation Drawer, só falta implementar o método `onNavDrawerItemSelected()`, o qual é chamado ao selecionar um item de menu.



MainActivity.java

```
public class MainActivity . . . {
    . . .
    @Override
    public void onNavDrawerItemSelected(NavigationDrawerFragment navigationDrawerFragment,
        int position) {
        List<NavDrawerMenuItem> list = NavDrawerMenuItem.getList();
        NavDrawerMenuItem selectedItem = list.get(position);
        // Seleciona a linha
        this.listAdapter.setSelected(position, true);
        toast("Clicou no item: " + getString(selectedItem.title));
    }
}
```

Neste código estamos chamando o método `setSelected(position,true)` para indicar que o item de menu está selecionado; sendo assim, o adapter vai desenhar o fundo cinza nesse item. Agora execute o projeto e selecione um item de menu. O resultado será um alerta (toast) com o nome do item selecionado.

13.9 Criando o menu overflow

Por padrão, o Android Studio cria um item de menu com o texto **Settings**. Vamos alterar esse arquivo de menu para criar um item chamado Sobre (About).



/res/menu/menu_main.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto" >
    <item
        android:id="@+id/action_about" android:title="@string/action_about"
        android:orderInCategory="100" app:showAsAction="never" />
</menu>
```

Adicione o texto do menu no arquivo */res/values/strings.xml*.



/res/values/strings.xml

```
<resources>
    . . .
    <string name="action_about">Sobre</string>
</resources>
```

E no código da activity vamos tratar o evento de Sobre (About).



MainActivity.java

```
...
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.action_about) {
        toast("Clicou no Sobre");
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

Ao clicar no menu **Sobre**, um toast será exibido (Figura 13.9). Posteriormente vamos melhorar esta tela e criar um `FragmentManager` com a descrição do projeto. Essa funcionalidade é bem comum nos aplicativos para que o usuário obtenha informações adicionais.

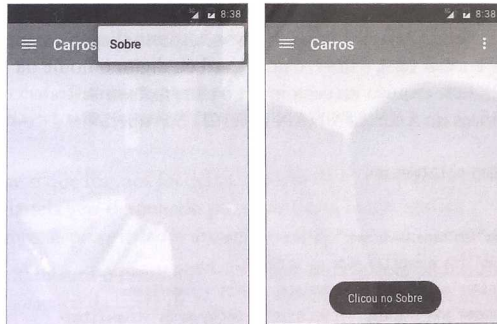


Figura 13.9 – Item de menu *Sobre* (About).

13.10 Navigation Drawer com Material Design

Criar o Navigation Drawer até que foi simples, o mais complicado vai ser customizá-lo para o Material Design, pois se você olhar o aplicativo do Gmail que segue as boas práticas o menu deslizante abre por cima da barra de status.

E, como podemos verificar, nosso aplicativo não está fazendo isso. Então vamos continuar a brincadeira, pois ainda não acabamos. Agora a coisa vai ficar um pouco complicada, mas espero que com o passo a passo demonstrado aqui você consiga implementar o Navigation Drawer com os padrões do Material Design.

Primeiramente, deixe o arquivo *styles.xml* conforme demonstrado a seguir. Veja que estou definindo dois temas, o *AppTheme* e *AppTheme.NavDrawer*.

 /res/values/styles.xml

```
<resources>
    <!-- Tema Base com configurações genéricas -->
    <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
        // Não mudei nada aqui
    </style>
    <!-- Tema para o Nav Drawer. -->
    <style name="AppTheme.NavDrawer" parent="AppTheme">
        <!-- Habilita o overlay mode na action bar -->
        <item name="windowActionBarOverlay">true</item>
    </style>
</resources>
```

Agora crie o arquivo */res/values-v21/styles.xml* específico para o Android 5.0 (API Level 21) ou superior. Para criar esses arquivos, costumo clicar no arquivo *styles.xml* que já existe e teclar **Ctrl+C** e **Ctrl+V**, e no wizard eu digito o nome da pasta que é */res/values-v21*. Esse arquivo vai customizar o tema *AppTheme.NavDrawer* com propriedades específicas do Android 5.0 (API Level 21) ou superior.

 /res/values-v21/styles.xml

```
<resources>
    <style name="AppTheme.NavDrawer" parent="AppTheme">
        <!-- Habilita o overlay mode na action bar -->
        <item name="android:windowTranslucentStatus">true</item>
        <item name="android:windowDrawsSystemBarBackgrounds">true</item>
        <item name="android:statusBarColor">@color/primary_dark</item>
    </style>
</resources>
```

Importante: para validar as funcionalidades referentes ao Material Design, teste no emulador ou dispositivo com Android 5.0 ou superior.

O objetivo de criar o tema `AppTheme.NavDrawer` é deixar a barra de sistema transparente. Como só queremos fazer isso na `MainActivity`, altere o arquivo `AndroidManifest.xml` conforme demonstrado a seguir. Com esta configuração, todas as outras activities que ainda vamos criar vão utilizar o tema padrão, somente a `MainActivity` vai utilizar o tema `AppTheme.NavDrawer`.

AndroidManifest.xml

```
<manifest . . . >
    <application . . . android:theme="@style/AppTheme" >
        <activity
            android:name=".activity.MainActivity" android:theme="@style/AppTheme.NavDrawer" . . . />
        </application>
    </manifest>
```

Se você executar o projeto agora, verá que o menu lateral já ocupou a tela inteira. Mas, por causa dessa alteração, a barra de status está transparente, e precisamos desenhar a cor dela manualmente. Isso é feito com uma única linha de código, conforme demonstrado a seguir:

MainActivity.java

```
. . .
protected void onCreate(Bundle savedInstanceState) {
    . . .
    // Cor de fundo da barra de status
    drawerLayout.setStatusBarBackground(R.color.primary_dark);
}
```

Basicamente o que fizemos foi deixar a barra de status transparente, para que o conteúdo da tela seja desenhado por cima dessa barra. Isso foi possível porque customizamos as propriedades do tema.

A figura 13.10 mostra o resultado. Veja que a `Toolbar` está deslocada para cima e invadindo o espaço da barra de status (system bar). Ao abrir o menu lateral, veja que ele abre por cima de tudo (inclusive da `Toolbar` e da status bar), o que está correto no Material Design.

O problema é que, quando o menu do Navigation Drawer estiver fechado, não queremos que a `Toolbar` seja deslocada para cima. Para resolver esse problema, o Google criou a classe `ScreenInsetsFrameLayout` que faz parte do aplicativo do Google I/O 2014, cujo código-fonte está disponível no GitHub: <https://github.com/google/iosched>.

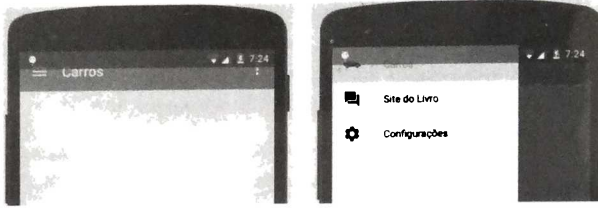


Figura 13.10 – Action Bar Overlay (transparente).

Eu copiei essa classe para a biblioteca android-utils, portanto basta adicioná-la no layout. Para manter o padrão, deixei a classe `ScrimInsetsFrameLayout` no mesmo pacote que estava no projeto do Google I/O. Para utilizar essa classe, adicione a propriedade `android:fitsSystemWindows="true"` na raiz do layout, que é o `DrawerLayout`, e envolva o fragment do Navigation Drawer lá no final do código-fonte com a classe `ScrimInsetsFrameLayout`.



/res/layout/activity_main.xml

```
<android.support.v4.widget.DrawerLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true">
    <LinearLayout ... >
        <!-- Aqui fica tudo igual -->
    </LinearLayout>
    <com.google.samples.apps.iosched.ui.widget.ScrimInsetsFrameLayout
        android:id="@+id/containerScrimInsets"
        android:layout_width="wrap_content" android:layout_height="match_parent"
        android:layout_gravity="start" android:elevation="8dp"
        android:fitsSystemWindows="true" app:insetForeground="#4000">
        <!-- Lista Menu Lateral -->
        <fragment
            android:id="@+id/nav_drawer_fragment"
            android:name="livroandroid.lib.fragment.NavigationDrawerFragment"
            android:layout_width="@dimen/navigation_drawer_width"
            android:layout_height="match_parent"
            android:layout_gravity="start" />
    </com.google.samples.apps.iosched.ui.widget.ScrimInsetsFrameLayout>
</android.support.v4.widget.DrawerLayout>
```

O atributo `android:fitsSystemWindows="true"` que foi colocado na raiz do layout faz com que o layout seja desenhado mais para baixo, encaixando-se na janela que o sistema utiliza para desenhar a aplicação. Ao fazer isso, o Navigation Drawer também vai abrir mais para baixo, porém no caso do menu queremos manter ele abrindo sobre a barra de sistema, pois esse é o padrão do Material Design. Justamente por isso, o fragment que controla o Navigation Drawer foi envolvido com a classe `ScreenInsetsFrameLayout` do Google, que por sua vez faz a mágica de desenhar o Navigation Drawer por cima de tudo. Esse é o pulo do gato!

Ao executar o projeto, o resultado deve ser como a figura 13.11.

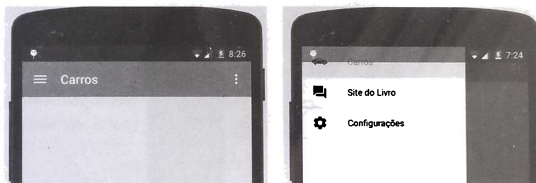


Figura 13.11 – Mágica com o Navigation Drawer.

13.11 Material Design no Navigation Drawer

Já fizemos o Navigation Drawer seguindo os padrões do Material Design, pois o menu lateral está abrindo sobre a barra de status. No entanto, se você comparar com alguns aplicativos do Google como o Gmail, verá que está faltando aquele cabeçalho em que podemos visualizar uma bonita imagem e geralmente informações do usuário.

Então vamos fazer isso para dar um acabamento profissional no aplicativo. Adicione a seguinte linha no arquivo de layout do menu lateral. Isso vai incluir um cabeçalho logo acima do `ListView`.

 `/res/layout/nav_drawer_listview.xml`

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/listViewContainer" android:orientation="vertical"
    android:layout_width="match_parent" android:layout_height="match_parent" >
    <!-- Cabeçalho para o Nav Drawer (Material Design) -->
    <include layout="@layout/nav_drawer_listview_header" />
    <ListView
        android:id="@+id/listView"
```



```

        android:layout_width="match_parent" android:layout_height="0dp"
        android:layout_weight="1" android:background="#ffffff" />
    <TextView . . .
        android:text="Livro Android - Todos os direitos reservados" />
</LinearLayout>

```

Feito isso, execute o aplicativo, e o resultado será como a figura 13.12. O arquivo *@layout/nav_drawer_listview_header* faz parte da biblioteca *android-utils*, e apresenta uma bonita imagem de fundo. Se você quiser, copie o código-fonte do arquivo e coloque no seu projeto.



Figura 13.12 – Cabeçalho no Navigation Drawer.

Para fechar o assunto com chave de ouro, vamos adicionar uma linha de código no método *getNavDrawerView()*, o qual é responsável por criar a view do menu lateral. O que vamos fazer é simplesmente preencher o layout do cabeçalho com a foto (logo) do aplicativo ou usuário, e algumas informações adicionais como nome e email.



MainActivity.java

```

public class MainActivity extends BaseActivity implements
    NavigationDrawerFragment.NavigationDrawerCallbacks {
    . . .
    @Override
    public NavigationDrawerFragment.NavDrawerListView getNavDrawerView(. . .) {
        View view = layoutInflater.inflate(R.layout.nav_drawer_listview, container, false);
        // Preenche o cabeçalho com a foto, nome e email.
        navigationDrawerFragment.setHeaderValues(view, R.id.listViewContainer,
            R.drawable.nav_drawer_header, R.drawable.ic_logo_user, R.string.nav_drawer_username,
            R.string.nav_drawer_email);
        return new NavigationDrawerFragment.NavDrawerListView(view,R.id.listView);
    }
    . . .
}

```

Dica: na página do Android Design, no tópico sobre o Navigation Drawer, podemos encontrar a especificação do layout para esse cabeçalho. As boas práticas de interface definem métricas de espaçamentos, alinhamentos, fontes etc.

Para o código compilar, copie a figura `R.drawable.ic_logo_user` (imagem do livrinho) para o seu projeto. Ela pode ser encontrada nos projetos de exemplo do livro. Os textos `R.string.nav_drawer_username` e `R.string.nav_drawer_email` precisam ser inseridos no arquivo `/res/values/strings.xml`.

 `/res/values/strings.xml`

```
<resources>
    . . .
    <string name="nav_drawer_username">Ricardo Lecheta</string>
    <string name="nav_drawer_email">rlecheta@gmail.com</string>
</resources>
```

Feito isso, execute o projeto de novo e finalmente teremos o Navigation Drawer seguindo as boas práticas do Material Design (Figura 13.13). No código eu deixei de forma estática a figura e os textos com o nome e email do usuário, mas acredito que com este exemplo você consiga evoluir a ideia para seu aplicativo e atualizar as informações dinamicamente caso seja necessário.



Figura 13.13 – Navigation Drawer com Material Design.

Lembre-se de que utilizamos a biblioteca `android-utls` para auxiliar na explicação e deixar o código digitado mais simples, pois ela encapsula boa parte do trabalho que tivemos para construir o Navigation Drawer. O fragment que você adicionou

no arquivo de layout foi baseado no exemplo do próprio Android Studio, no wizard de template do Navigation Drawer. Eu apenas melhorei a ideia, e deixei o componente um pouco mais customizável. Quando achar necessário, estude o código-fonte desse fragment para melhorar o seu aprendizado.

13.12 Criando os fragments do projeto

Para finalizar o capítulo, vamos criar os fragments que serão adicionados na tela quando o usuário selecionar as opções do menu.

Faça isso com o wizard **New > Fragment > Fragment (Blank)** e no campo do nome do fragment digite `CarrosFragment` e no layout digite `fragment_carros`. O Android Studio vai criar os arquivos com um monte de código, mas pelo menos ele já criou tudo em seu devido lugar. Apague todo o código gerado, e deixe simples conforme demonstrado a seguir:

 `CarrosFragment.java`

```
package br.com.livroandroid.carros.fragments;

public class CarrosFragment extends BaseFragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_carros, container, false);
        return view;
    }
}
```

Importante: no wizard não selecione as opções `Include fragment factory methods` e `include interface callbacks`, pois se fizer isso ele vai gerar código demais. De qualquer forma, depois de gerar as classes com o wizard, sempre deixe-as igual ao código-fonte demonstrado no livro.

Lembre-se de que todos os fragments do projeto devem herdar de `BaseFragment` que criamos anteriormente. Esse é o fragment que vai mostrar a lista de carros. (clássicos, esportivos e luxo). Por enquanto, o fragment não faz nada, e no arquivo de layout temos apenas um texto qualquer.

 /res/layout/fragment_carros.xml

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Listá de Carros Aqui" android:layout_gravity="center" />
</FrameLayout>
```

Vamos criar também o fragment para abrir a página do livro com um WebView.

 SiteLivroFragment.java

```
package br.com.livroandroid.carros.fragments;
public class SiteLivroFragment extends BaseFragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_site_livro, container, false);
        return view;
    }
}
```

 /res/layout/fragment_site_livro.xml

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Página do Livro Aqui" android:layout_gravity="center" />
</FrameLayout>
```

Ainda não vamos fazer nada com o item de menu configurações, pois vamos estudar essa parte no capítulo 18, sobre persistência.

Depois de criar os fragments, vamos adicioná-los dinamicamente no layout da activity principal. Para isso, altere o método `onNavDrawerItemSelected(...)` da `MainActivity`. Por enquanto esse método mostra um toast com o título do menu selecionado, mas desta vez, ao selecionar um item, vamos adicionar o fragment no layout.



MainActivity.java

```

...
import android.support.v4.app.Fragment;
public class MainActivity extends BaseActivity implements
    NavigationDrawerFragment.NavigationDrawerCallbacks {
    ...
    @Override
    public void onNavDrawerItemSelected(NavigationDrawerFragment navigationDrawerFragment,
        int position) {
        List<NavDrawerMenuItem> list = NavDrawerMenuItem.getList();
        NavDrawerMenuItem selectedItem = list.get(position);
        // Seleciona a linha
        this.listAdapter.setSelected(position, true);
        if (position == 0) {
            replaceFragment(new CarrosFragment());
        } else if (position == 1) {
            replaceFragment(new SiteLivreFragment());
        } else if (position == 2) {
            toast("Configurações");
        } else {
            Log.e("livroandroid", "Item de menu inválido");
        }
    }
    // Adiciona o fragment no centro da tela
    private void replaceFragment(Fragment frag) {
        getSupportFragmentManager().beginTransaction().replace(R.id.nav_drawer_container,
            frag, "TAG").commit();
    }
}

```

Veja que criei o método `replaceFragmetn(frag)` para facilitar a leitura do código, e preste atenção pois você precisa importar a classe da biblioteca de compatibilidade, que é `android.support.v4.app.Fragment`. Veja que a `FragmentManager` está fazendo `replace()` e não `add()`, pois sempre queremos substituir o fragment do conteúdo por outro.

Ao terminar essas configurações, execute o projeto novamente, e o resultado deve ser como a figura 13.14, que mostra o fragment dos carros. Na parte da esquerda da figura podemos ver o menu aberto, e na direita o fragment que foi adicionado com o conteúdo ao clicar no menu **Carros**.

O item de menu **Site do Livro** vai adicionar seu respectivo fragment na tela, e o item **Configurações** vai mostrar um simples toast por enquanto.

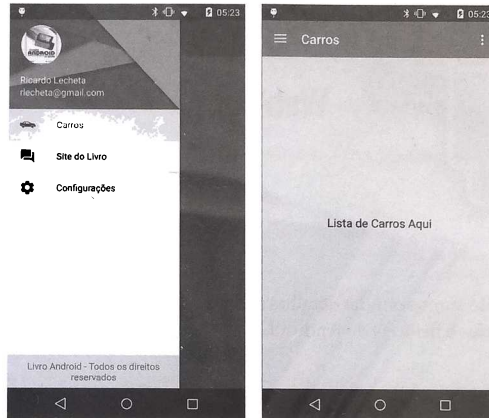


Figura 13.14 – Fragments em ação.

13.3 Links úteis

Neste capítulo, criamos o projeto dos carros, e talvez este tenha sido o passo mais difícil, pois geralmente o complicado é começar.

Aprendemos a utilizar o padrão de navegação do Navigation Drawer, e para continuar os seus estudos separei alguns links da documentação oficial.

- **Android Developer – Patterns – Navigation Drawer**
<https://developer.android.com/design/patterns/navigation-drawer.html>
- **Material Design – Navigation Drawer**
<http://www.google.com/design/spec/patterns/navigation-drawer.html>
- **Android Training – Overlaying the Action Bar**
<https://developer.android.com/training/basics/actionbar/overlaying.html>



CAPÍTULO 14

WebView

Neste capítulo, vamos estudar detalhes sobre a classe `WebView`, um dos componentes mais utilizados e flexíveis do Android.

14.1 Fragment com WebView

Vamos continuar o desenvolvimento do projeto dos carros e o próximo objetivo é utilizar um `WebView` na classe `SiteLivroFragment` para abrir esta página da internet.

<http://www.livroandroid.com.br/sobre.htm>

Atualize o código do arquivo de layout conforme demonstrado a seguir:

```
 /res/layout/fragment_site_livro.xml

<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <WebView android:id="@+id/webview"
        android:layout_width="match_parent" android:layout_height="match_parent" />
    <ProgressBar android:id="@+id/progress"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="center" />
</FrameLayout>
```

Observe que o layout raiz é um `FrameLayout` para deixar o `ProgressBar` por cima do `WebView`. No código do fragment basta mostrar a página no `WebView`.

```
 SiteLivroFragment.java

package br.com.livroandroid.carros.fragments;
...
public class SiteLivroFragment extends BaseFragment {
```

```

private static final String URL_SOBRE = "http://www.livroandroid.com.br/sobre.htm";
private WebView webView;
private ProgressBar progress;
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.fragment_site_livro, container, false);
    webView = (WebView) view.findViewById(R.id.webview);
    progress = (ProgressBar) view.findViewById(R.id.progress);
    setWebViewClient(webView);
    // Carrega a página
    webView.loadUrl(URL_SOBRE);
    return view;
}
private void setWebViewClient(WebView webView) {
    webView.setWebViewClient(new WebViewClient() {
        @Override
        public void onPageStarted(WebView webView, String url, Bitmap favicon) {
            super.onPageStarted(webview, url, favicon);
            // Liga o progress
            progress.setVisibility(View.VISIBLE);
        }
        @Override
        public void onPageFinished(WebView webView, String url) {
            // Desliga o progress
            progress.setVisibility(View.INVISIBLE);
        }
    });
}
}

```

Para o WebView acessar a internet, adicione esta permissão no *AndroidManifest.xml*:

```

AndroidManifest.xml AndroidManifest.xml

<manifest . . .>
    <uses-permission android:name="android.permission.INTERNET" />
    <application . . . />
</manifest>

```

Feito isso, execute o projeto, e o resultado deve ser como a figura 14.1. Acredito que seja simples entender esse exemplo, pois já estudamos o *WebView* no capítulo 7, sobre views. Sendo assim, nos próximos tópicos vamos explorar algumas funcionalidades mais avançadas.



Figura 14.1 – Página do livro com WebView.

14.2 Swipe to Refresh

Um padrão de design muito conhecido em aplicativos mobile é o Swipe to Refresh, o qual permite atualizar os dados da lista quando o usuário faz o gesto de rolagem para baixo.

Para fazer o Swipe to Refresh, envolva o `WebView` com o layout `SwipeRefreshLayout`.

```

<?xml version="1.0" encoding="utf-8"?>
<FrameLayout . . .>
    <android.support.v4.widget.SwipeRefreshLayout
        android:id="@+id/swipeToRefresh"
        android:layout_width="match_parent" android:layout_height="match_parent">
        <WebView . . . />
    </android.support.v4.widget.SwipeRefreshLayout>
    <ProgressBar . . . />
</FrameLayout>

```

O `SwipeRefreshLayout` é um gerenciador de layout que deve envolver outra view (apenas uma), para que automaticamente seja adicionado o suporte à atualização da view por meio do gesto de rolagem para baixo. No código a única tarefa necessária é configurar o listener com o método `setOnRefreshListener(listener)`. Feito isso, quando o gesto de rolagem for feito, o método `onRefresh()` será chamado – momento em

que podemos chamar o método `webview.reload()` para atualizar a página. Observe que no método `onPageFinished()` do `WebViewClient` estamos encerrando a animação do `SwipeRefreshLayout`.



SitELivroFragment.java

```

...
public class SitELivroFragment extends BaseFragment {
    ...
    protected SwipeRefreshLayout swipeLayout;

    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        ...
        // Swipe to Refresh
        swipeLayout = (SwipeRefreshLayout) view.findViewById(R.id.swipeToRefresh);
        swipeLayout.setOnRefreshListener(OnRefreshListener());
        swipeLayout.setColorSchemeResources(
            R.color.refresh_progress_1,
            R.color.refresh_progress_2,
            R.color.refresh_progress_3);

        return view;
    }

    private SwipeRefreshLayout.OnRefreshListener OnRefreshListener() {
        return new SwipeRefreshLayout.OnRefreshListener() {
            @Override
            public void onRefresh() {
                webview.reload(); // Atualiza a página
            }
        };
    }

    private void setWebViewClient(WebView webview) {
        webview.setWebViewClient(new WebViewClient() {
            ...
            @Override
            public void onPageFinished(WebView webview, String url) {
                // Desliga o progress
                progress.setVisibility(View.INVISIBLE);
                // Termina a animação do Swipe to Refresh
                swipeLayout.setRefreshing(false);
            }
        });
    }
}

```

Para o código compilar, crie as cores da animação do `SwipeRefreshLayout`, que é uma bolinha girando que fica alternando entre as cores que você definir.

 /res/values/colors.xml

```
<resources>
    . . .
    <color name="refresh_progress_1">#33B5E5</color> <!-- azul -->
    <color name="refresh_progress_2">#99CC00</color> <!-- verde -->
    <color name="refresh_progress_3">#CC0000</color> <!-- vermelho -->
</resources>
```

A figura 14.2 mostra o resultado com a animação do `Swipe to Refresh`, a fim de atualizar a página.



Figura 14.2 – Animação do `Swipe to Refresh`.

Dica: para fazer o gesto `Pull to Refresh`, toque na lista e segure, depois arraste para baixo e solte. É o mesmo gesto utilizado em aplicativos nativos como o Gmail.

14.3 Interceptando requisições no `WebView`

Uma funcionalidade muito comum em aplicativos que utilizam `WebView` é interceptar as URLs para executar alguma lógica no aplicativo.

No caso do aplicativo dos carros, o `WebView` abriu a seguinte página:

<http://www.livroandroid.com.br/sobre.htm>

Nesta página existe um link **Mais Informações** que está chamando a própria página `sobre.htm` novamente. Nosso objetivo é interceptar o clique nesse link para mostrar um alerta com algumas informações adicionais de forma nativa no aplicativo. Isso

é simples de fazer, basta sobrescrever o método `shouldOverrideUrlLoading(view,url)` da classe `WebViewClient`.

SiteLivreFragment.java

```
public class SiteLivreFragment extends BaseFragment {
    ...
    private void setWebViewClient(WebView webview) {
        webview.setWebViewClient(new WebViewClient() {
            ...
            @Override
            public boolean shouldOverrideUrlLoading(WebView view, String url) {
                Log.d("livroandroid", "webview url: " + url);
                if (url != null && url.endsWith("sobre.htm")) {
                    toast("Clicou em Mais Informações");
                    // Retorna true para informar que interceptamos o evento
                    return true;
                }
                return super.shouldOverrideUrlLoading(view, url);
            }
        });
    }
}
```

14.4 Mostrando alertas com o `FragmentManager`

No tópico anterior, interceptamos o clique no link **Mais Informações**. Portanto, ao executar o projeto e clicar nesse link, o aplicativo vai mostrar um toast.

Mas em vez do toast queremos mostrar um alerta (dialog) com informações sobre o aplicativo e com links para o usuário clicar e ver mais informações. Para mostrar o alerta, vamos utilizar a classe `DialogFragment`, pois é a maneira recomendada de mostrar alertas segundo as boas práticas do Android. A classe `DialogFragment` é um fragment usado para mostrar alertas customizados. A vantagem de utilizar um fragment em vez do convencional `AlertDialog` é que podemos customizar a view do fragment, e o fragment também faz parte do ciclo de vida da activity. Para mostrar ou esconder um `FragmentManager`, podemos utilizar a API da `FragmentManager` conforme já estudamos, portanto todos os conceitos sobre fragments continuam sendo aplicados. O simples fato de herdar de `DialogFragment` fará com que a view desse fragment seja exibida como um alerta.

O objetivo é mostrar no alerta um texto com informações do aplicativo, portanto adicione as seguintes mensagens no arquivo `/res/values/strings.xml`:

```
 /res/values/strings.xml

<resources>
    . . .
    <!-- Sobre -->
    <string name="about_dialog_text">
<![CDATA[
<b>Aplicativo dos Carros 2015</b><br>
Versão %s<br><br>
<a href="http://www.livroandroid.com.br">http://livroandroid.com.br</a><br><br>
<a href="http://www.facebook.com/ricardolecheta">http://facebook.com/ricardolecheta</a><br><br>
<a href="https://plus.google.com/+RicardoLecheta">https://plus.google.com/+RicardoLecheta</a>
]]>
    </string>
    <string name="about_dialog_title">Livro Android</string>
    <string name="ok">OK</string>
</resources>
```

O layout da janela do alerta será um simples `TextView` que mostra o texto `@string/about_dialog_text`.

```
 /res/layout/dialog_about.xml

<TextView xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@android:id/text1"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="24dp" android:text="@string/about_dialog_text" />
```

Para implementar a classe do fragment, temos duas opções. Podemos sobrescrever o método `onCreateView()` como qualquer outro fragment e retornar a view do alerta, ou sobrescrever o método `onCreateDialog(Bundle)` para criar um alerta customizado utilizando o `AlertDialog`. Se você sobrescrever o método `onCreateView()`, terá de retornar uma view com pelo menos um botão que será usado para fechar a janela, e também um título para o alerta. Eu particularmente gosto de sobrescrever o método `onCreateDialog(Bundle)` para casos simples como esse, pois podemos utilizar um `AlertDialog` que já insere um título na janela e cria um botão OK para fechá-la.

```
 AboutDialog.java
```

```
package br.com.livroandroid.carros.fragments;
import android.app.AlertDialog;
```

```

import android.app.Dialog;
import android.support.v4.app.DialogFragment;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentTransaction;
import livroandroid.lib.utils.AndroidUtils;
...
public class AboutDialog extends DialogFragment {
    // Método utilitário para mostrar o dialog
    public static void showAbout(android.support.v4.app.FragmentManager fm) {
        FragmentTransaction ft = fm.beginTransaction();
        Fragment prev = fm.findFragmentByTag("dialog_about");
        if (prev != null) {
            ft.remove(prev);
        }
        ft.addToBackStack(null);
        new AboutDialog().show(ft, "dialog_about");
    }
    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        // Cria o HTML com o texto de about
        SpannableStringBuilder aboutBody = new SpannableStringBuilder();
        String versionName = AndroidUtils.getVersionName(getActivity());
        aboutBody.append(Html.fromHtml(getString(R.string.about_dialog_text, versionName)));
        // Infla o layout
        LayoutInflater inflater = LayoutInflater.from(getActivity());
        TextView view = (TextView) inflater.inflate(R.layout.dialog_about, null);
        view.setText(aboutBody);
        view.setMovementMethod(new LinkMovementMethod());
        // Cria o dialog customizado
        return new AlertDialog.Builder(getActivity())
            .setTitle(R.string.about_dialog_title)
            .setView(view)
            .setPositiveButton(R.string.ok,
                new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int whichButton) {
                        dialog.dismiss();
                    }
                }
            )
            .create();
    }
}

```

Veja que estamos utilizando a classe `android.support.v4.app.DialogFragment` do pacote de compatibilidade e não a classe nativa `android.app.DialogFragment`. Também estou utilizando a classe `livroandroid.lib.utils.AndroidUtils` da biblioteca `android-utils` para retornar o código da versão do aplicativo, que é o atributo `versionName` cadastrado no *app/build.gradle*. Depois de criar o `FragmentManager`, vamos atualizar o código da classe `SiteLivroFragment`; assim, quando o usuário clicar no link **Mais Informações** da página, o evento será interceptado para abrir o alerta.

SiteLivroFragment.java

```
@Override
public boolean shouldOverrideUrlLoading(WebView view, String url) {
    Log.d("livroandroid", "webview url: " + url);
    if (url != null && url.endsWith("sobre.htm")) {
        AboutDialog.showAbout(getFragmentManager());
    }
    return super.shouldOverrideUrlLoading(view, url);
}
```

O resultado podemos conferir na figura 14.3. Veja que o alerta mostrou o título **Livro Android** e o botão **OK** que fecha a janela. Se você clicar no botão voltar, a janela também será fechada, pois o fragment foi adicionado na back stack. Observe que no código-fonte utilizei a classe `Html.fromHtml(string)` para mostrar um código HTML dentro do `TextView`. Inclusive o usuário pode clicar nos links para visualizar a página no browser.

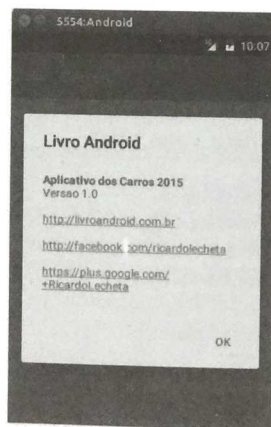


Figura 14.3 – *FragmentManager*.

Aproveitando que fizemos a classe `AboutDialog`, vamos ajustar a classe `MainActivity` do projeto, pois no capítulo anterior deixamos o menu `/res/menu/menu_main.xml` com um item de menu chamado **Sobre**, que é mostrado nas opções do menu overflow da action bar, ou seja, a `Toolbar`.

 **MainActivity.java**

```
...
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.action_about) { // Mostra o dialog com informações do aplicativo
        AboutDialog.showAbout(getSupportFragmentManager());
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

14.5 Executando JavaScript

Para que o `WebView` execute qualquer script na página, precisamos habilitar o JavaScript, conforme demonstrado a seguir:

```
WebSettings settings = webView.getSettings();
settings.setJavaScriptEnabled(true);
```

Feito isso, qualquer código JavaScript vai funcionar no `WebView` e inclusive você poderá injetar códigos JavaScript dentro do `WebView`, como por exemplo:

```
webView.loadUrl("javascript:alert('Olá!');");
```

Eu não recomendo esse tipo de técnica, mas já houve casos em que precisei alterar o CSS da página web (à qual eu não tinha acesso ao servidor) e fiz isso injetando um JavaScript na página.

14.6 Comunicação do JavaScript com a classe Android

Outra funcionalidade interessante e perigosa é a possibilidade de a página web executar métodos que estão dentro da activity ou fragment que contém o `WebView`.

Para isso, basta chamar o método `addJavascriptInterface(object, nomeObjeto)` do `WebView` para configurar o objeto que vai conter os métodos que o JavaScript pode chamar. O primeiro parâmetro é a instância do objeto que contém os métodos, e o segundo parâmetro é o nome do objeto que o JavaScript vai utilizar.

O código a seguir mostra como expor um objeto "LivroAndroid" para a página.



SiteLivroFragment.java

```

...
@Override
public void onCreateView(LayoutInflater inflater, ViewGroup container,
    Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.fragment_site_livro, container, false);
    webView = (WebView) view.findViewById(R.id.webview);
    ...
    configJavaScript();
    return view;
}

private void configJavaScript() {
    WebSettings settings = webView.getSettings();
    // Ativa o JavaScript na página
    settings.setJavaScriptEnabled(true);
    // Publica a interface para o JavaScript
    webView.addJavaScriptInterface(new LivroAndroidInterface(), "LivroAndroid");
}

class LivroAndroidInterface {
    @JavascriptInterface
    public void sobre() {
        toast("Clicou na figura do livro!");
    }
}

...
}

```

Dica: a partir do Android 4.3 (Jelly Bean), somente os métodos que forem anotados com `@JavascriptInterface` serão expostos ao `WebView`. Isso é feito por questões de segurança, pois antigamente a página podia chamar qualquer método.

Isso significa que na página web podemos usar o objeto "LivroAndroid" para executar os métodos que estão definidos na classe `LivroAndroidInterface`, pois é isso que essa linha de código faz.

```
webView.addJavaScriptInterface(new LivroAndroidInterface(), "LivroAndroid");
```

Se você olhar o código-fonte da página *sobre.htm* na página do livro, verá que ele está chamando o seguinte JavaScript ao clicar na figura:

```
<script type="text/javascript">
    function sobre() {
        // Função JavaScript que chama o método na Activity
        LivroAndroid.sobre();
    }
</script>
```

Para testar a brincadeira, execute o projeto novamente e clique na figura do livrinho. Isso vai chamar o método `sobre()` dentro da classe `SiteLivroFragment`.

14.7 Mostrando código HTML no WebView

Eu não vou prolongar muito o assunto sobre `WebView`, até porque recomendo que sempre que possível toda a interface seja feita de forma nativa. Mas uma última dica de que talvez você possa precisar é sobre criar um código HTML em `String` e mandar abrir no `WebView`.

Isso pode ser feito chamando o método `loadData()`:

```
webview.loadData("<html><body> HTML aqui </body></html>", "text/html", "UTF-8");
```

14.8 Links úteis

Neste capítulo, aprendemos detalhes importantes sobre o `WebView` e como utilizar a classe `FragmentManager`. Para continuar seus estudos, separei alguns links.

- **Android Developers – Building Web Apps in WebView**

<http://developer.android.com/guide/webapps/webview.html>

- **WebView API**

<http://developer.android.com/reference/android/webkit/WebView.html>

- **Android Developers Blog – Using DialogFragments**

<http://android-developers.blogspot.com.br/2012/05/using-dialogfragments.html>

- **FragmentManager API**

<http://developer.android.com/reference/android/app/FragmentManager.html>



CAPÍTULO 15

RecyclerView e tabs

Neste capítulo, vamos criar a lista de carros e aproveitar para revisar os conceitos do Material Design. O objetivo é guiá-lo passo a passo no desenvolvimento de um aplicativo Android. Deixaremos toda a estrutura da lista pronta para quando formos buscar os carros do web service.

15.1 Criando as classes de domínio

Para continuar o projeto dos carros, vamos criar a classe Carro, mas antes crie o pacote `br.com.livroandroid.carros.domain` com o objetivo de armazenar as classes de domínio. Uma maneira de fazer isso é clicar com o botão direito no pacote `br.com.livroandroid.carros` e usar o menu **New > Package**; depois digite `domain` no nome do pacote.

Feito isso, crie as classes Carro e CarroService neste pacote, conforme a figura 15.1.



Carro.java

```
package br.com.livroandroid.carros.domain;
import java.io.Serializable;
public class Carro implements Serializable {
    private static final long serialVersionUID = 6601006766832473959L;
    public long id;
    public String tipo;
    public String nome;
    public String desc;
    public String urlFoto;
    public String urlInfo;
    public String urlVideo;
    public String latitude;
    public String longitude;
    @Override
```

```

    public String toString() {
        return "Carro{" + "nome=" + nome + '\'' + '\'';
    }
}

```

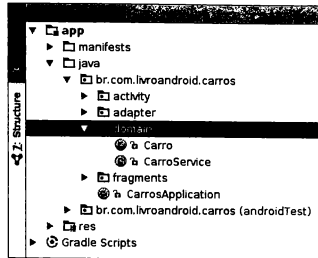


Figura 15.1 – Pacote com as classes de domínio.

Dica: para gerar o método `toString()` de uma classe, utilize o menu **Code > Generate > toString()** e escolha os atributos que você deseja imprimir. Esse mesmo wizard apresenta opções para gerar construtores e métodos do tipo getters e setters e também pode ser invocado pela tecla de atalho `Ctrl+Insert`.

A classe `CarroService` contém o método `getCarros(context)` para criar uma lista de carros de forma fixa em memória. Futuramente essa lista será criada a partir de um web service.



CarroService.java

```

package br.com.livroandroid.carros.domain;

...

public class CarroService {
    public static List<Carro> getCarros(Context context) {
        List<Carro> carros = new ArrayList<Carro>();
        for (int i = 0; i < 20; i++) {
            Carro c = new Carro();
            c.nome = "Carro " + i;
            c.desc = "Desc " + i;
            c.urlFoto = "http://www.livroandroid.com.br/livro/carros/esportivos/Ferrari_FF.png";
            carros.add(c);
        }
        return carros;
    }
}

```

15.2 Criando a lista de carros

Para criar a lista de carros, utilizaremos o RecyclerView e CardView que já estudamos. Portanto vamos ser rápidos, pois queremos entrar logo nas partes mais interessantes do projeto.

Para começar, declare a dependência das bibliotecas, inclusive do Picasso (<https://github.com/square/picasso>), que é uma excelente biblioteca utilizada para o download de imagens.



app/build.gradle

```
...
compile 'com.android.support:recyclerview-v7:22.1.0'
compile 'com.android.support:cardview-v7:22.1.0'
compile 'com.squareup.picasso:picasso:2.5.2'
```

Como nosso objetivo é criar a lista de carros, vamos começar pelo layout do adapter, que tem um TextView para o nome e um ImageView para a foto.



/res/layout/adapter_carro.xml

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v7.widget.CardView xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:id="@+id/card_view"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:layout_margin="6dp" android:clickable="true"
    card_view:cardBackgroundColor="@color/white" card_view:cardCornerRadius="2dp"
    card_view:cardElevation="6dp"
    android:foreground="?attr/selectableItemBackground">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="?android:attr/listPreferredItemHeight"
        android:gravity="center_vertical" android:orientation="horizontal">
        <FrameLayout
            android:layout_width="wrap_content" android:layout_height="wrap_content">
            <!-- Foto do carro -->
            <ImageView android:id="@+id/img"
                android:layout_width="116dp" android:layout_height="50dp" />
            <!-- Barra de progresso enquanto carrega a foto -->
            <ProgressBar android:id="@+id/progressImg"
                style="@android:style/Widget.ProgressBar.Small"
                android:layout_width="wrap_content" android:layout_height="wrap_content"
```

```

        android:layout_gravity="center|center_vertical" android:layout_marginRight="6dp"
        android:gravity="center|center_vertical" android:visibility="invisible" />
    </FrameLayout>
    <!-- Nome carro -->
    <TextView android:id="@+id/text"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:layout_marginLeft="10dp" android:textColor="@color/primary" />
</LinearLayout>
</android.support.v7.widget.CardView>

```

O atributo `android:foreground="?attr/selectableItemBackground"` é para o `CardView` criar o efeito de ripple no Android 5.0 ou superior, e vai manter a compatibilidade simulando algum efeito de toque nas versões antigas. Lembrando que o efeito de ripple ou qualquer outro é muito importante para fornecer o feedback ao toque (Touch Feedback) para o usuário. Por padrão, o efeito de ripple apresenta uma cor cinza claro. Para definir a cor, customizamos o tema com a propriedade `colorControlHighlight`.



/res/values/styles.xml

```

. . .
<!-- Cor para o efeito ripple -->
<item name="colorControlHighlight">@color/control_highlight</item>

```



/res/values/colors.xml

```

. . .
<color name="control_highlight">#B3E5FC</color>

```

Se você utilizou a mesma cor que está neste livro, o efeito de ripple é azul claro. Lembrando que todas as cores são baseadas na paleta de cores do Material Design.

<http://www.google.com/design/spec/style/color.html>

Atenção: cuidado para não deixar a cor `colorControlHighlight` igual à cor primária do aplicativo, pois às vezes o efeito ripple pode não aparecer se o componente utiliza a mesma cor. Mais para frente, vamos fazer a tab (clássicos, esportivos, luxo) ser da cor primária do aplicativo, pois é o padrão do Material Design. Neste caso, se a cor do `colorControlHighlight` for igual, você não verá o efeito de ripple ao selecionar uma tab, portanto tome cuidado. Esse probleminha aconteceu comigo e levei tempo para descobrir o motivo de o efeito não aparecer.

A seguir, podemos ver o código da classe de adapter que vai utilizar o layout `/res/layout/adapter_carro.xml` que acabamos de criar.



CarroAdapter.java

```
package br.com.livroandroid.carros.adapter;

...

public class CarroAdapter extends RecyclerView.Adapter<CarroAdapter.CarrosViewHolder> {
    protected static final String TAG = "livroandroid";
    private final List<Carro> carros;
    private final Context context;
    private CarroOnClickListener carroOnClickListener;
    public CarroAdapter(Context context, List<Carro> carros, CarroOnClickListener
        carroOnClickListener) {
        this.context = context;
        this.carros = carros;
        this.carroOnClickListener = carroOnClickListener;
    }
    @Override
    public int getItemCount() { return this.carros != null ? this.carros.size() : 0; }
    @Override
    public CarrosViewHolder onCreateViewHolder(ViewGroup viewGroup, int viewType) {
        View view = LayoutInflater.from(context).inflate(R.layout.adapter_carro, viewGroup, false);
        CarrosViewHolder holder = new CarrosViewHolder(view);
        return holder;
    }
    @Override
    public void onBindViewHolder(final CarrosViewHolder holder, final int position) {
        // Atualiza a view
        Carro c = carros.get(position);
        holder.tNome.setText(c.nome);
        holder.progress.setVisibility(View.VISIBLE);
        // Faz o download da foto e mostra o ProgressBar
        Picasso.with(context).load(c.urlFoto).fit().into(holder.img,
            new com.squareup.picasso.Callback() {
                @Override
                public void onSuccess() {
                    holder.progress.setVisibility(View.GONE); // download ok
                }
                @Override
                public void onError() {
                    holder.progress.setVisibility(View.GONE);
                }
            });
    }
};
```

```

        // Click
        if (carroOnClickListener != null) {
            holder.itemView.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    // A variável position é final
                    carroOnClickListener.onClickCarro(holder.itemView, position);
                }
            });
        }
    }

    public interface CarroOnClickListener {
        public void onClickCarro(View view, int idx);
    }

    // ViewHolder com as views
    public static class CarrosViewHolder extends RecyclerView.ViewHolder {
        public TextView tNome;
        ImageView img;
        ProgressBar progress;
        CardView cardView;
        public CarrosViewHolder(View view) {
            super(view);
            // Cria as views para salvar no ViewHolder
            tNome = (TextView) view.findViewById(R.id.text);
            img = (ImageView) view.findViewById(R.id.img);
            progress = (ProgressBar) view.findViewById(R.id.progressImg);
            cardView = (CardView) view.findViewById(R.id.card_view);
        }
    }
}

```

Dica: para carregar a foto do carro, estamos utilizando a biblioteca Picasso, que facilita o download e inclusive faz cache das imagens. No código usamos um `ProgressBar` para fazer a animação durante o download e a interface `com.squareup.picasso.Callback` para receber o evento quando o carregamento da imagem terminar. Outra opção seria utilizar o Picasso apenas com uma imagem temporária (placeholder) durante o download, mas o exemplo com `ProgressBar` é mais difícil e estamos fazendo assim por motivos de aprendizado.

A esta altura do livro, o código do adapter deve ser tranquilo para você, portanto vamos continuar. Adicione um `RecyclerView` no layout do fragment que vai listar os carros.

 /res/layout/fragment_carros.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="14dp">
    <android.support.v7.widget.RecyclerView android:id="@+id/recyclerView"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:cacheColorHint="@android:color/transparent"
        android:clipToPadding="false"
        android:divider="@null" android:dividerHeight="0dp"
        android:listSelector="@android:color/transparent"
        android:scrollbarStyle="outsideOverlay" android:scrollbars="vertical" />
</FrameLayout>
```

Na classe CarrosFragment vamos utilizar o método CarroService.getCarros(context) para obter a lista de carros a fim de preencher o adapter do RecyclerView.

 CarrosFragment.java

```
package br.com.livroandroid.carros.fragments;
...
public class CarrosFragment extends BaseFragment {
    protected RecyclerView recyclerView;
    private List<Carro> carros;
    private LinearLayoutManager mLayoutManager;
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_carros, container, false);
        recyclerView = (RecyclerView) view.findViewById(R.id.recyclerView);
        mLayoutManager = new LinearLayoutManager(getActivity());
        recyclerView.setLayoutManager(mLayoutManager);
        recyclerView.setItemAnimator(new DefaultItemAnimator());
        recyclerView.setHasFixedSize(true);
        return view;
    }
    @Override
    public void onActivityCreated(@Nullable Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);
        taskCarros();
    }
    private void taskCarros() {
        // Busca os carros
    }
}
```

```

        this.carros = CarroService.getCarros(getContext());
        // Atualiza a lista
        recyclerView.setAdapter(new CarroAdapter(getContext(), carros, onClickCarro()));
    }

    private CarroAdapter.CarroOnClickListener onClickCarro() {
        return new CarroAdapter.CarroOnClickListener() {
            @Override
            public void onClickCarro(View view, int idx) {
                Carro c = carros.get(idx);
                Toast.makeText(getContext(), "Carro: " + c.nome, Toast.LENGTH_SHORT).show();
            }
        };
    }
}

```

O método `onActivityCreated(bundle)` do fragment geralmente é um bom lugar para iniciar a lógica da tela, como por exemplo buscar dados do web service ou do banco de dados. Nele estou chamando o método `taskCarros()` para criar a lista de carros e preencher a lista. Por enquanto, o método `taskCarros()` não faz muita coisa, mas já estamos preparando a estrutura do código para quando formos utilizar threads (`AsyncTask`) para buscar os carros do web service.

Concluídos esses passos, execute o projeto, e o resultado deve ser como a figura 15.2. Ao selecionar um carro da lista, o seu nome é exibido em um toast. Conforme explicado anteriormente, adicionamos o item `"?attr/selectableItemBackground"` no layout do adapter, assim temos o efeito de ripple ao tocar no `CardView`.

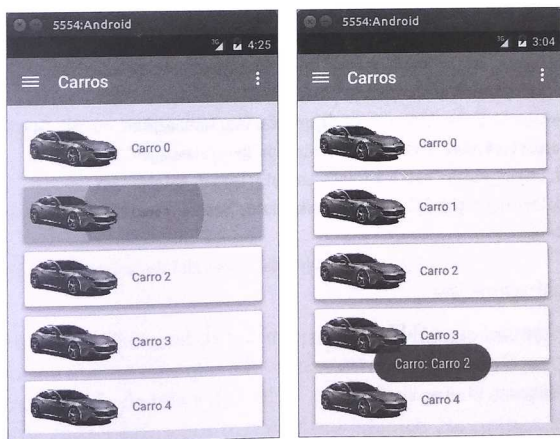


Figura 15.2 – Lista de carros.

15.3 Tabs e ViewPager

Já criamos a lista de carros, mas no aplicativo precisamos de três tabs, que são: clássicos, esportivos e luxo.

Conforme explicado no capítulo 5, sobre action bar, a navegação por tabs utilizando a action bar foi descontinuada (deprecated). Porém, no Google I/O 2015, o Google anunciou a biblioteca *Android Design Support Library* que contém classes e componentes para auxiliar na construção de aplicativos com Material Design. Dentre esses componentes foi criado o `TabLayout`, utilizado para criar as tabs.

Para utilizar essa nova biblioteca, declare a dependência no *app/build.gradle*.

 **app/build.gradle**

```
...
compile 'com.android.support:design:22.2.0'
```

Então vamos colocar a mão na massa. Crie um novo fragment chamado `CarrosTabFragment` com este arquivo de layout, que contém o `TabLayout` com `ViewPager`:

 **/res/layout/fragment_carros_tab.xml**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical">
    <!-- As tabs são apenas para visualizar -->
    <android.support.design.widget.TabLayout
        android:id="@+id/tabLayout"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:background="@color/primary" />
    <!-- O ViewPager é preenchido com os fragments via TabsAdapter -->
    <android.support.v4.view.ViewPager android:id="@+id/viewpager"
        android:layout_width="match_parent" android:layout_height="0px"
        android:layout_weight="1" android:background="@android:color/white" />
</LinearLayout>
```

 **CarrosTabFragment.java**

```
package br.com.livroandroid.carros.fragments;
import android.support.design.widget.TabLayout;
import android.support.v4.view.ViewPager;
```

```

public class CarrosTabFragment extends BaseFragment
    implements TabLayout.OnTabSelectedListener {
    private ViewPager mViewPager;
    private TabLayout tabLayout;
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_carros_tab, container, false);
        // ViewPager
        mViewPager = (ViewPager) view.findViewById(R.id.viewpager);
        mViewPager.setOffscreenPageLimit(2);
        mViewPager.setAdapter(new TabsAdapter(getContext(), getChildFragmentManager()));
        // Tabs
        tabLayout = (TabLayout) view.findViewById(R.id.tabLayout);
        int cor = getContext().getResources().getColor(R.color.white);
        // Cor branca no texto (o fundo azul foi definido no layout)
        tabLayout.setTabTextColors(cor, cor);
        // Adiciona as tabs.
        tabLayout.addTab(tabLayout.newTab().setText(R.string.classicos));
        tabLayout.addTab(tabLayout.newTab().setText(R.string.esportivos));
        tabLayout.addTab(tabLayout.newTab().setText(R.string.luxo));
        // Listener para tratar eventos de clique na tab
        tabLayout.setOnTabSelectedListener(this);
        // Se mudar o ViewPager atualiza a tab selecionada
        mViewPager.addOnPageChangeListener(new TabLayout.TabLayoutOnPageChangeListener(tabLayout));
        return view;
    }
    @Override
    public void onTabSelected(TabLayout.Tab tab) {
        // Se alterar a tab, atualiza o ViewPager
        mViewPager.setCurrentItem(tab.getPosition());
    }
    @Override
    public void onTabUnselected(TabLayout.Tab tab) { }
    @Override
    public void onTabReselected(TabLayout.Tab tab) { }
}

```

Como podemos ver, o `TabLayout` da biblioteca `Android Design` é simples e funciona da mesma forma que no exemplo feito com tabs na action bar. Porém, quem controla todo o conteúdo da tela e das tabs é o `ViewPager`, pois é ele que vai fornecer o conteúdo de cada página por meio de um adapter. Na prática, a tab apenas mostra a página que está selecionada no `ViewPager`.

Observe que esta linha de código faz com que o `ViewPager` mantenha vivas sempre duas tabs a mais do que ele está visualizando. Assim, todas as três tabs ficam sempre em memória. Como são poucas informações, no caso do aplicativo dos carros isso fica bom.

```
viewPager.setOffscreenPageLimit(2);
```

Para prosseguir, crie a classe do adapter que vai retornar os fragments para o `ViewPager`. Como as três tabs vão mostrar a lista de carros, usaremos sempre o fragment `CarrosFragment` que contém a lista, mas vamos passar um argumento pelo `Bundle` para indicar o tipo de carro que será mostrado na-lista.



TabsAdapter.java

```
package br.com.livroandroid.carros.adapter;
...
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentPagerAdapter;
public class TabsAdapter extends FragmentPagerAdapter {
    private Context context;
    public TabsAdapter(Context context, FragmentManager fm) {
        super(fm);
        this.context = context;
    }
    @Override
    public int getCount() { return 3; }
    @Override
    public CharSequence getPageTitle(int position) {
        if (position == 0) {
            return context.getString(R.string.classicos);
        } else if (position == 1) {
            return context.getString(R.string.esportivos);
        }
        return context.getString(R.string.luxo);
    }
    @Override
    public Fragment getItem(int position) {
        Bundle args = new Bundle();
        if (position == 0) {
            args.putString("tipo", "classicos");
        } else if (position == 1) {
            args.putString("tipo", "esportivos");
        }
    }
}
```

```

    } else {
        args.putString("tipo", "luxo");
    }
    Fragment f = new CarrosFragment();
    f.setArguments(args);
    return f;
}
}

```

Nota: lembre-se de que no código sempre vamos utilizar as versões de compatibilidade v4 dos fragments. Tenha atenção ao fazer os imports.

Para o código compilar, adicione os textos das tabs no arquivo *strings.xml*.

```

 /res/values/strings.xml

<resources>
    . . .
    <!-- Tab Carros -->
    <string name="classicos">Clássicos</string>
    <string name="esportivos">Esportivos</string>
    <string name="luxo">Luxo</string>
</resources>

```

Neste momento, se você executar o projeto as tabs e o ViewPager já vão funcionar, porém as três tabs vão mostrar a mesma lista de carros. Portanto, vamos aprimorar a lógica da classe CarrosFragment para ler o tipo do carro dos argumentos e criar uma lista conforme esse tipo.

```

 CarrosFragment.java

. . .
public class CarrosFragment extends BaseFragment {
    . . .
    private String tipo;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        if (getArguments() != null) {
            this.tipo = getArguments().getString("tipo");
        }
    }
}

```

```

    . . .
    private void taskCarros() {
        this.carros = CarroService.getCarros(getContext(), tipo);
        . . .
    }
}

```

Para o código compilar, adicione o parâmetro `tipo` no método `getCarros(context, tipo)`.

CarroService.java

```

    . . .
    public static List<Carro> getCarros(Context context, String tipo) {
        for (int i = 0; i < 20; i++) {
            Carro c = new Carro();
            c.nome = "Carro " + tipo + ": " + i; // Nome dinâmico conforme o tipo
            . . .
        }
    }

```

Note que só é preciso alterar uma linha nesse código, apenas para o nome do carro conter o tipo selecionado. Por enquanto, a lista de carros continuará estática para facilitar a construção da navegação do projeto.

Para finalizar a configuração, precisamos alterar o código que trata o evento do Navigation Drawer, para mostrar o fragment com as **Tags** ao selecionar a opção **Carros**. Altere o método `onNavigationItemSelectedListener(...)` para mostrar a classe `CarrosTabFragment` em vez da `CarrosFragment`, conforme demonstrado a seguir.

MainActivity.java

```

    . . .
    @Override
    public void onNavigationItemSelectedListener(NavigationDrawerFragment navigationDrawerFragment,
        int position) {
        . . .
        if (position == 0) {
            replaceFragment(new CarrosTabFragment());
        } else if (position == 1) { . . . }
        } else if (position == 2) { . . . }
    }
}

```

O resultado deve ser como a figura 15.3, que mostra a tab **Esportivos** selecionada.



Figura 15.3 – Tabs.

15.4 Navegação de telas

Ao seleccionar um carro na lista, o aplicativo deve navegar para a activity que mostra a tela de detalhes do carro, então vamos lá. Serei breve nas explicações, pois já estudamos isso no livro, portanto aproveite para revisar os conceitos e praticar.

Crie as classes `CarroActivity` e `CarroFragment` conforme demonstrado a seguir. Note que a palavra "carro" no nome das classes está no singular. Para a activity, recomendo utilizar o wizard **New > Activity**, pois o wizard, além de criar os arquivos, também configura a activity no arquivo *AndroidManifest.xml*.

A classe `CarroActivity` receberá o objeto do carro como parâmetro pelo `Bundle` e envia esse objeto para o fragment por meio do método `setCarro(carro)`, que por sua vez é responsável tanto pelo layout quanto pelo conteúdo da tela.



CarroActivity.java

```
package br.com.livroandroid.carros.activity;

public class CarroActivity extends BaseActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_carro);
    }
}
```



```

        // Configura a Toolbar como a action bar
        setUpToolbar();
        // Atualiza o carro no fragment
        CarroFragment cf = (CarroFragment) getSupportFragmentManager()
            .findFragmentById(R.id.CarroFragment);
        Carro c = (Carro) getIntent().getSerializableExtra("carro");
        cf.setCarro(c);
        // Título da toolbar e botão up navigation
        getSupportActionBar().setTitle(c.nome);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    }
}

```

O layout da activity apenas insere o fragment `CarroFragment` que cuidará do conteúdo da tela. Foi dado um id para o fragment, pois assim podemos obtê-lo com o método `findFragmentById(id)`. Tenha atenção aos layouts das activities; todas elas precisam incluir a Toolbar, pois lembre-se de que desativamos a action bar padrão.

 /res/layout/activity_carro.xml

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" >
    <include layout="@layout/include_toolbar" />
    <fragment android:id="@+id/CarroFragment"
        class="br.com.livroandroid.carros.fragments.CarroFragment"
        android:layout_width="match_parent" android:layout_height="match_parent"
        tools:layout="@layout/fragment_carro" />
</LinearLayout>

```

No arquivo *AndroidManifest.xml* configure a activity `CarroActivity` para que sua activity mãe seja a `MainActivity`, assim podemos utilizar o up navigation.

 AndroidManifest.xml

```

<application . . .>
    <activity android:name=".activity.CarroActivity"
        android:label="@string/title_activity_carro"
        android:parentActivityName=".activity.MainActivity" >
        <meta-data android:name="android.support.PARENT_ACTIVITY"
            android:value=".activity.MainActivity" />
    </activity>
</application>

```

Capítulo 15 ■ RecyclerView e tabs

Para o código compilar, crie o fragment.



CarroFragment.java

```
package br.com.livroandroid.carros.fragments;

public class CarroFragment extends BaseFragment {
    private Carro carro;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_carro, container, false);
        return view;
    }

    // Método público chamado pela activity para atualizar os dados do carro
    public void setCarro(Carro carro) {
        if (carro != null) {
            this.carro = carro;
            setTextColor(R.id.tDesc, carro.desc);
            final ImageView imgView = (ImageView) getView().findViewById(R.id.img);
            Picasso.with(getContext()).load(carro.urlFoto).fit().into(imgView);
        }
    }
}
```



/res/layout/fragment_carro.java

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" android:padding="12dp">
    <android.support.v7.widget.CardView android:id="@+id/card_view"
        android:layout_width="match_parent" android:layout_height="match_parent"
        android:clickable="true" card_view:cardBackgroundColor="@color/white"
        card_view:cardCornerRadius="2dp" card_view:cardElevation="6dp">
    <LinearLayout
        android:layout_width="match_parent" android:layout_height="match_parent"
        android:orientation="vertical" android:padding="2dp">
    <ImageView android:id="@+id/img"
        android:layout_width="@dimen/foto_carro_width"
        android:layout_height="@dimen/foto_carro_height"
        android:layout_gravity="center" android:scaleType="center" />
    <View android:id="@+id/text"
        android:layout_width="match_parent" android:layout_height="1dp">
```

```

        android:layout_marginBottom="6dp" android:layout_marginTop="6dp" />
    <ScrollView
        android:layout_width="match_parent" android:layout_height="wrap_content">
        <TextView android:id="@+id/tDesc"
            android:layout_width="match_parent" android:layout_height="match_parent"
            android:layout_weight="1" android:textColor="@color/primary_dark"
            android:layout_margin="8dp"/>
    </ScrollView>
</LinearLayout>
</android.support.v7.widget.CardView>
</LinearLayout>

```

 /res/values/dimens.xml

```

<resources>
    . . .
    <!-- Foto do carro -->
    <dimen name="foto_carro_width">260dp</dimen>
    <dimen name="foto_carro_height">110dp</dimen>
</resources>

```

Para fazer a navegação de telas, altere o código que trata o evento da lista.

 CarrosFragment.java

```

. . .
public class CarrosFragment extends BaseFragment {
    private CarroAdapter.CarroOnClickListener onClickCarro() {
        . . .
        public void onClickCarro(View view, int idx) {
            Carro c = carros.get(idx);
            Intent intent = new Intent(getContext(), CarroActivity.class);
            intent.putExtra("carro", c);
            startActivity(intent);
        }
    }
}

```

Nota: a classe Carro implementa a interface de marcação `java.io.Serializable`, por isso podemos passá-la como parâmetro pelo Bundle utilizando o método `putExtra(string, serializable)`. Basicamente isso faz com que o objeto seja serializado (convertido para bytes) para depois na outra activity fazer o processo inverso (bytes para objeto). Vale ressaltar que na documentação do Android é recomendado utilizar a interface `android.os.Parcelable`, pois é mais performática.

Eu particularmente nunca senti nenhuma demora ao passar objetos como `Serializable`, portanto faço assim no meu dia a dia e explico deste jeito. Mas, caso você precise passar por parâmetro um objeto realmente grande ou uma lista de objetos e perceba algum atraso (delay) na navegação de telas, vale a pena procurar como implementar a interface `Parcelable`.

O layout do fragment tem o `ImageView` para a foto e um `TextView` para a descrição do carro. Foi utilizado um `ScrollView` para fazer rolagem no caso de telas pequenas, pois a descrição do carro pode exceder a altura disponível da tela. Ao executar o aplicativo e selecionar um carro na lista, o objeto do carro será passado como parâmetro pelo `Bundle`, e a `activity` `CarroActivity` vai ler esse objeto para configurar o conteúdo do fragment. Lembre-se de que a `activity` é responsável pela navegação e o fragment pelo conteúdo. O resultado da navegação de telas deve ser como a figura 15.4:

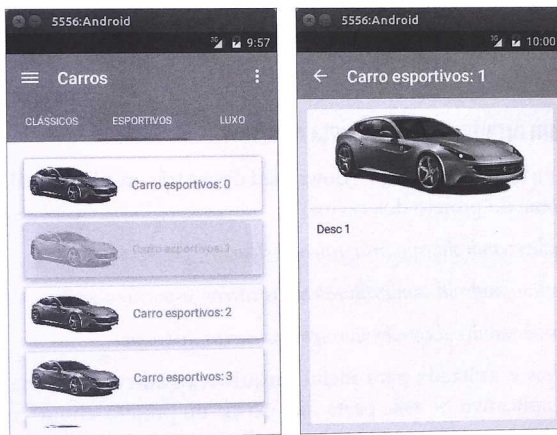


Figura 15.4 – Navegação de telas.

15.5 Links úteis

Neste capítulo, implementamos a lista de carros e revisamos importantes conceitos do desenvolvimento de aplicativos para Android. No próximo capítulo, vamos buscar a lista de carros de um web service. Para continuar seus estudos, separei um link.

- **Android Training – Creating Lists and Cards**

<https://developer.android.com/training/material/lists-cards.html>



CAPÍTULO 16

Parser de XML, JSON e testes unitários

Geralmente, web services retornam os dados no formato XML ou JSON. Neste capítulo, vamos aprender como ler esse tipo de informação.

16.1 Lendo um arquivo local da pasta /res/raw

Para começar a brincadeira, faça o download destes três arquivos XML e insira-os na pasta /res/raw do projeto dos carros:

http://www.livroandroid.com.br/livro/carros/carros_classicos.xml

http://www.livroandroid.com.br/livro/carros/carros_esportivos.xml

http://www.livroandroid.com.br/livro/carros/carros_luxo.xml

A pasta /res/raw é utilizada para incluir arquivos estáticos que são compilados junto com o aplicativo. Se essa pasta não existir no projeto, clique com o botão direito na pasta /res e utilize o menu **New Directory** para criá-la. Para sua validação, o resultado deve ser como a figura 16.1.

Sempre que um arquivo for inserido na pasta /res/raw, será gerado um recurso na classe R, por exemplo: R.raw.nome_arquivo. Para ler esse arquivo, podemos utilizar o método getResources().openRawResource(R.raw.x), que retorna uma InputStream.

```
Resources resources = context.getResources();  
InputStream in = resources.openRawResource(R.raw.nome_arquivo);
```

A interface java.io.InputStream do Java define um fluxo de dados para leitura, e com ela podemos ler facilmente o arquivo utilizando a classe java.io.FileInputStream. Bem, a partir daqui é Java puro, portanto, se você já trabalhou com arquivos em Java vai se sentir em casa.

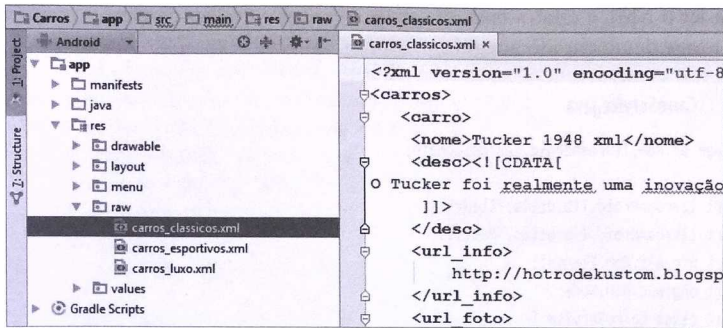


Figura 16.1 – Pasta /res/raw com arquivos XML.

Para facilitar seus estudos, criei a classe `livroandroid.lib.utils.FileUtils`, portanto para ler um arquivo da pasta `/res/raw` em formato String basta utilizar este código:

```
String xml = FileUtils.readRawFileString(context, R.raw.carros_classicos, "UTF-8");
```

Neste caso, o arquivo `/res/raw/carros_classicos.xml` será lido e convertido para String.

16.2 Parser de XML

Com o objetivo de criar a lista de carros, vamos aprender a ler o arquivo XML que colocamos no projeto, o qual contém a seguinte estrutura:

```

📄 /res/raw/carros_esportivos.xml

<?xml version="1.0" encoding="utf-8"?>
<carros>
  <carro>
    <nome> </nome>
    <desc> </desc>
    <url_info> </url_info>
    <url_foto> </url_foto>
    <url_video> </url_video>
    <latitude> </latitude>
    <longitude> </longitude>
  </carro>
  . . .
  <carro> outros carros aqui </carro>
</carros>

```

Para ler o XML e criar a lista de carros, atualize o código da classe `CarroService` conforme demonstrado a seguir.

CarroService.java

```
package br.com.livroandroid.carros.domain;

...
import livroandroid.lib.utils.FileUtils;
import livroandroid.lib.utils.XMLUtils;
import org.w3c.dom.Element;
import org.w3c.dom.Node;

public class CarroService {

    private static final boolean LOG_ON = false;
    private static final String TAG = "CarroService";

    public static List<Carro> getCarros(Context context, String tipo) {
        try {
            String xml = readFile(context, tipo);
            List<Carro> carros = parserXML(context, xml);
            return carros;
        } catch (Exception e) {
            // TODO explicar exception
            Log.e(TAG, "Erro ao ler os carros: " + e.getMessage(), e);
            return null;
        }
    }

    // Faz a leitura do arquivo que está na pasta /res/raw
    private static String readFile(Context context, String tipo) throws IOException {
        if ("classicos".equals(tipo)) {
            return FileUtils.readRawFileString(context, R.raw.carros_classicos, "UTF-8");
        } else if ("esportivos".equals(tipo)) {
            return FileUtils.readRawFileString(context, R.raw.carros_esportivos, "UTF-8");
        }
        return FileUtils.readRawFileString(context, R.raw.carros_luxo, "UTF-8");
    }

    // Faz o parser do XML e cria a lista de carros
    private static List<Carro> parserXML(Context context, String xml) {
        List<Carro> carros = new ArrayList<Carro>();
        Element root = XMLUtils.getRoot(xml, "UTF-8");
        // Le todas as tags <carro>
        List<Node> nodeCarros = XMLUtils.getChildren(root, "carro");
        // Insere cada carro na lista
        for (Node node : nodeCarros) {
            Carro c = new Carro();
```

```

// Lê as informações de cada carro
c.nome = XMLUtils.getText(node, "nome");
c.desc = XMLUtils.getText(node, "desc");
c.urlFoto = XMLUtils.getText(node, "url_foto");
c.urlInfo = XMLUtils.getText(node, "url_info");
c.urlVideo = XMLUtils.getText(node, "url_video");
c.latitude = XMLUtils.getText(node, "latitude");
c.longitude = XMLUtils.getText(node, "longitude");
if (LOG_ON) {
    Log.d(TAG, "Carro " + c.nome + " > " + c.urlFoto);
}
carros.add(c);
}
if (LOG_ON) {
    Log.d(TAG, carros.size() + " encontrados.");
}
return carros;
}
}

```

O método `CarroService.getCarros(context, string)` recebe o tipo do carro desejado, que pode ser: esportivos, luxo ou clássicos. Baseado no tipo do carro, o arquivo XML correto é lido da pasta `/res/raw`. O retorno é uma `String` no formato XML, então é feito o parser e logo depois é criada a lista de carros. Ao executar o projeto novamente, o resultado deve ser como a figura 16.2, que mostra a lista de carros criada com base no arquivo XML.

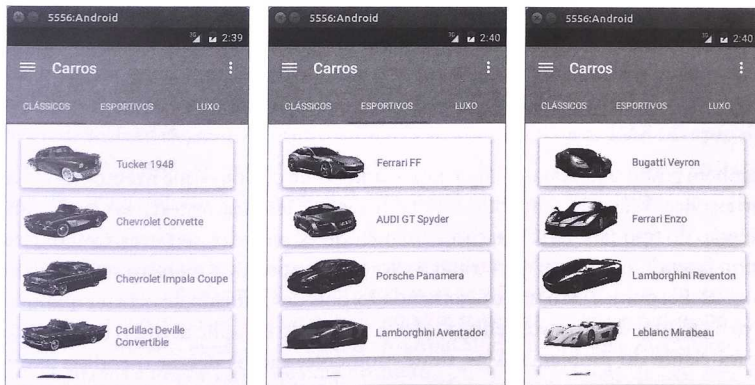


Figura 16.2 – Lista de carros do web service.

Como no layout do adapter `/res/layout/adapter_carro.xml` foi adicionado um `ProgressBar` e utilizamos a biblioteca Picasso para fazer o download da imagem, o `ProgressBar` é mostrado durante o download, conforme a figura 16.3. Isso acontece porque no arquivo XML os dados dos carros estão reais, inclusive a URL das fotos. No lado direito da figura, podemos ver que a navegação de telas continua funcionando, pois a única diferença é que todas as informações vieram do XML.

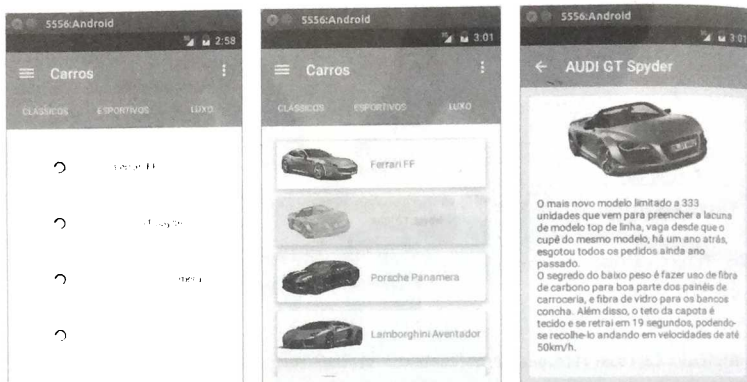


Figura 16.3 – `ProgressBar` e navegação de telas.

Para finalizar este tópico, gostaria de ressaltar que o parser de XML pode ser feito com SAX ou DOM. O SAX percorre o XML passo a passo, utiliza menos recursos e por isso é mais performático. O DOM cria uma árvore em memória que representa a estrutura do XML e permite acessar os elementos de forma rápida. Se o XML for grande e você precisar de desempenho, recomenda-se fazer o parser por SAX, mas eu costumo utilizar o DOM pela simplicidade e nunca tive problemas. A classe `XMLUtils` utilizada dentro do código da classe `CarroService` facilita justamente a leitura do XML por DOM.

Também gostaria de ressaltar algo importante sobre o código que fizemos na classe `CarroService`. Veja que o método `FileUtils.readRawFileString(context, raw)` lança uma exceção do tipo `java.io.IOException`, e no método `CarroService.getCarros(context, tipo)` estou fazendo o `try/catch` para tratar a exceção e imprimir a mensagem de erro no `LogCat`. Na prática, isso não é recomendado, pois dessa forma, se ocorrer qualquer erro na leitura do arquivo, nenhum alerta será dado no aplicativo e o usuário não ficará ciente do problema. Mas, neste momento, vamos deixar o código assim, pois o objetivo foi explicar como ler um arquivo da pasta `/res/raw` e fazer o parser do XML. No capítulo 17, sobre web services, vamos organizar melhor o código e propagar as exceções adequadamente.

16.3 Parser de JSON

Outro formato de arquivo que recentemente ficou bastante popular por sua simplicidade é o JSON, o qual vem substituindo o XML como forma de integração de sistemas.

JSON (JavaScript Object Notation) é uma estrutura de dados que representa um objeto em JavaScript. Seu formato é simples, conforme demonstrado a seguir:

```
{
  "carros": {
    "carro": [
      {
        "nome": "Carro 1",
        "desc": "Desc Carro 1",
        "url_info": "url aqui",
        "url_foto": "url foto aqui",
        "url_video": "url video aqui",
        "latitude": "latitude aqui",
        "longitude": "longitude aqui"
      },
      { // Próximo carro aqui. }
    ]
  }
}
```

No site do livro existem estes arquivos JSON para cada tipo de carro:

http://www.livroandroid.com.br/livro/carros/carros_classicos.json

http://www.livroandroid.com.br/livro/carros/carros_esportivos.json

http://www.livroandroid.com.br/livro/carros/carros_luxo.json

Faça o download desses três arquivos JSON e insira-os na pasta `/res/raw` do projeto dos carros. É importante que você apague os arquivos XML que estavam na pasta para não gerar conflitos na classe R. A figura 16.4 mostra o resultado.

Nota: ao copiar os arquivos `.json` para a pasta `/res/raw`, certifique-se de apagar os arquivos `.xml`, pois a classe R não leva em consideração a extensão dos arquivos e neste caso o processo de compilação poderia se perder. Inclusive recomendo que você utilize o menu `Build > Clean` do Android Studio para limpar os arquivos compilados a fim de evitar qualquer conflito.

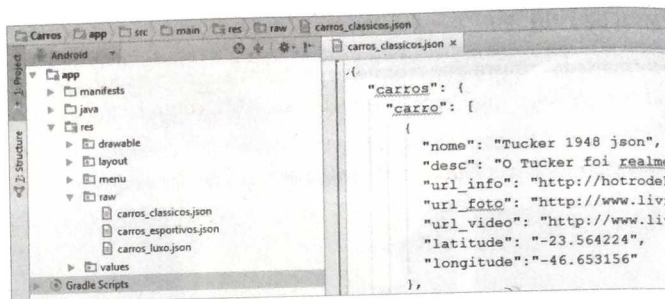


Figura 16.4 – Pasta /res/raw com arquivos JSON.

Depois de copiar os arquivos `.json` para a pasta `/res/raw`, atualize o código da classe `CarroService` para fazer o parser de JSON. Basicamente vamos criar o método `parserJSON(context,json)` para criar a lista de carros.



CarroService.java

```
package br.com.livroandroid.carros.domain;

...
import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;
public class CarroService {
    private static final boolean LOG_ON = false;
    private static final String TAG = "CarroService";
    public static List<Carro> getCarros(Context context, String tipo) {
        try {
            String json = readFile(context, tipo);
            List<Carro> carros = parserJSON(context, json);
            return carros;
        } catch (Exception e) {
            Log.e(TAG, "Erro ao ler os carros: " + e.getMessage(), e);
            return null;
        }
    }
    private static String readFile(Context context, String tipo) throws IOException {
        // Mesma coisa aqui, apenas lê os arquivos da pasta /res/raw
    }
    private static List<Carro> parserJSON(Context context, String json) throws IOException {
        List<Carro> carros = new ArrayList<Carro>();
        try {
            JSONObject root = new JSONObject(json);
```

```

JSONObject obj = root.getJSONObject("carros");
JSONArray jsonCarros = obj.getJSONArray("carro");
// Insere cada carro na lista
for (int i = 0; i < jsonCarros.length(); i++) {
    JSONObject jsonCarro = jsonCarros.getJSONObject(i);
    Carro c = new Carro();
    // Lê as informações de cada carro
    c.nome = jsonCarro.optString("nome");
    c.desc = jsonCarro.optString("desc");
    c.urlFoto = jsonCarro.optString("url_foto");
    c.urlInfo = jsonCarro.optString("url_info");
    c.urlVideo = jsonCarro.optString("url_video");
    c.latitude = jsonCarro.optString("latitude");
    c.longitude = jsonCarro.optString("longitude");
    if (LOG_ON) {
        Log.d(TAG, "Carro " + c.nome + " > " + c.urlFoto);
    }
    carros.add(c);
}
if (LOG_ON) {
    Log.d(TAG, carros.size() + " encontrados.");
}
} catch (JSONException e) {
    throw new IOException(e.getMessage(), e);
}
return carros;
}
}

```

Pronto, isso é tudo! Acredito que se você der uma leve estudada no código verá que fazer o parser de JSON é bem simples, e a classe `org.json.JSONObject` facilita bastante esse trabalho. Depois dessa alteração, o aplicativo dos carros deve continuar funcionando normalmente, pois mudamos apenas a classe que faz a busca dos carros e parser das informações. Se você é do tipo que precisa ter certeza para acreditar, altere o nome de um carro no arquivo JSON que está no projeto, e esse nome deverá aparecer na lista.

Nota: se você teve problemas para compilar o projeto, pode ser por causa de alguma sujeira que ficou nos arquivos compilados em razão de o arquivo XML ter o mesmo nome dos arquivos JSON. Se isso aconteceu com você, utilize o menu `Build > Clean` do Android Studio para limpar os arquivos compilados a fim de evitar qualquer conflito. Depois faça `Build > Rebuild Project` e execute o projeto novamente.

16.4 Testes unitários no Android

Uma das metodologias de desenvolvimento mais adotadas em diversas empresas é o TDD (Test Driven Development), que traduzindo é o desenvolvimento orientado a testes. O TDD consiste em uma boa prática de programação, que força o desenvolvedor a pensar no que vai fazer antes de sair programando. Enfim, esse não é um livro sobre TDD ou metodologias, portanto o que vou demonstrar agora é como executar um teste unitário no Android Studio.

Neste capítulo, implementamos o método `CarroService.getCarros(context, tipo)`, que recebe o tipo do carro desejado e faz o parser do XML ou JSON. Na verdade, já está tudo feito e funcionando, então, apenas para constar: seguindo as boas práticas do TDD, o que fizemos foi errado. O TDD prega que primeiro de tudo deve-se escrever o teste unitário. Somente depois que o teste tiver sido feito, o desenvolver fará a implementação do código, com o objetivo de passar no teste, ou seja, obter a famosa cor verde.

Contudo, como eu disse, este não é um livro sobre metodologias ou testes, então vamos logo para a prática. No Android Studio, crie uma classe chamada `CarroServiceTest` e coloque-a dentro do local (`androidTest`), conforme a figura 16.5.

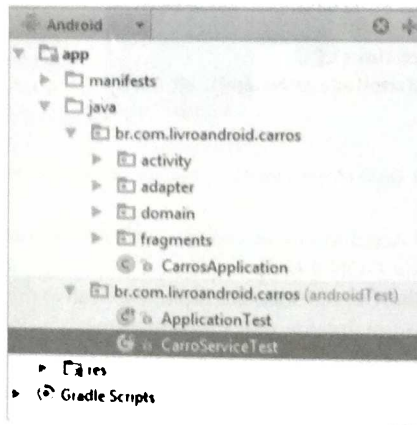


Figura 16.5 - Classe de testes.



CarroServiceTest.java

```

package br.com.livroandroid.carros;
import android.test.AndroidTestCase;
import java.io.IOException;
import java.util.List;
import br.com.livroandroid.carros.domain.Carro;
import br.com.livroandroid.carros.domain.CarroService;
public class CarroServiceTest extends AndroidTestCase {
    public void testGetCarros() throws IOException {
        List<Carro> carros = CarroService.getCarros(getContext(), "esportivos");
        assertNotNull(carros);
        // Precisa retornar dez carros esportivos.
        assertTrue(carros.size() == 10);
        // Valida as informações do 1º carro
        Carro c0 = carros.get(0);
        assertEquals("Ferrari FF", c0.nome);
        assertEquals("44.532218", c0.latitude);
        assertEquals("10.864019", c0.longitude);
        // Valida as informações do último carro
        Carro c9 = carros.get(9);
        assertEquals("MERCEDES-BENZ C63 AMG", c9.nome);
        assertEquals("-23.564224", c9.latitude);
        assertEquals("-46.653156", c9.longitude);
    }
}

```

A classe de teste unitário deve ser filha de `AndroidTestCase`, que por sua vez é filha da classe `TestCase` do popular framework JUnit (<http://junit.org/>). Essa classe permite executar um teste unitário no Android facilmente, pois a única coisa de que precisamos é o objeto `android.app.Context`, o qual pode ser obtido pelo método `getContext()`. Com o teste em mãos, clique com o botão direito no arquivo ou no centro do editor e selecione a opção **Run > CarroServiceTest**. Se estiver com o cursor dentro de um método do código, o Android Studio vai mostrar a opção para executar o método, como por exemplo: **Run > testGetCarros()**.

Enfim, se você já conhece como funciona o framework JUnit, nada será novidade para você. Caso contrário, recomendo procurar uma literatura adicional. Lembre-se de que fizemos o teste no final do capítulo, pois meu objetivo foi apenas demonstrar como criar um teste unitário no Android Studio (Figura 16.6). Lembre-se que caso o TDD estivesse sendo usado na prática, esse teste seria a primeira coisa a ser feita, mas isso é assunto para outro tipo de livro.

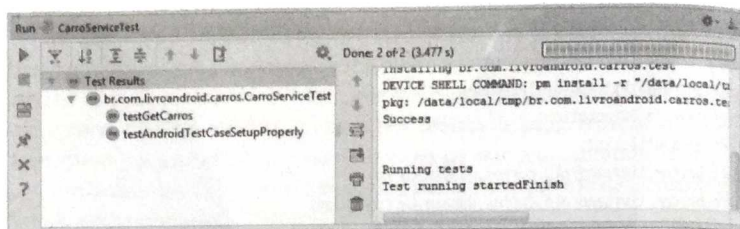


Figura 16.6 – Resultado dos testes.

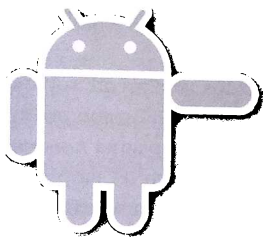
16.5 Mais informações

Este foi um breve capítulo sobre como fazer parser de arquivos nos formatos XML e JSON.

O que mostrei aqui foi uma maneira básica de como isso pode ser feito, mas provavelmente você vai encontrar por aí outras maneiras de fazer a mesma coisa. Por exemplo, existem bibliotecas bem populares, como o **Gson** (<https://code.google.com/p/google-gson/>), que permite converter um objeto para JSON e vice-versa, portanto seria possível fazer o parser em uma única linha.

Eu particularmente faço meus projetos exatamente como mostrei neste capítulo e tenho plena convicção de que consultar os web services e parser de XML/JSON são a parte mais simples do desenvolvimento. O complicado e às vezes demorado é criar interfaces gráficas bonitas que agradem o usuário.

Mas cada caso é diferente, e você deverá adaptar o que você está aprendendo ao seu dia a dia e aos requisitos de cada projeto.



CAPÍTULO 17

Web services

No capítulo anterior, aprendemos a fazer parser de arquivos XML e JSON.

Neste capítulo, vamos continuar o desenvolvimento do projeto e buscar a lista de carros de um web service.

17.1 Introdução

Uma dúvida comum de quem está iniciando com mobile ou desenvolvimento de sistemas é: como faço para me conectar ao banco de dados do servidor e buscar ou atualizar informações nesse banco de dados?

A resposta é: o aplicativo não deve conectar-se ao banco de dados do servidor.

A principal razão para isso é por questões de segurança dos dados e outra é por padronização de acesso, uma vez que web services facilitam a interoperabilidade entre diversas plataformas, pois não importa a linguagem de programação que o servidor ou cliente utilizam.

Para o mobile buscar informações do servidor, é necessário utilizar um web service; portanto, no servidor deve ser criado este web service em qualquer plataforma ou linguagem (Java, .NET, PHP, Ruby, Python). A responsabilidade do web service é acessar o banco de dados, processar as informações e retornar os dados no formato XML ou JSON para o mobile/cliente ler os dados.

Existem muitas maneiras de criar web services, mas isso está fora do escopo deste livro, pois estamos falando de mobile e não de servidor. Apenas para você complementar seus estudos, vou citar alguns temas para você pesquisar, caso ache necessário.

1. **WSDL (Web Services Description Language)** – É um formato de serviço escrito em XML, cujo tráfego de dados é feito via HTTP utilizando o protocolo SOAP (Simple Object Access Protocol). Esta é a maneira clássica de criar web services. Atualmente, esse modelo está sendo evitado no mundo mobile,

pois o protocolo SOAP é um grande XML que trafega pela rede, e pode causar lentidão, principalmente na conexão 3G.

2. **REST** – Formato leve de web services que são criados geralmente sobre o protocolo HTTP, utilizando os métodos GET, DELETE, POST e PUT como padronização de acesso. Por exemplo, se você fizer um GET na página */usuarios* pode ser retornada uma lista de usuários. Se você fizer um GET na página */usuario/1*, o usuário com o id=1 é retornado. Mas se você fizer um DELETE na página */usuario/1*, esse usuário será deletado. Outro exemplo é se fizermos um POST na página */usuario*, que neste caso serve para inserir um novo usuário. Então o padrão REST utiliza os próprios métodos do protocolo HTTP para criar a lógica necessária para o web service. No REST o formato de retorno mais comum é o JSON.
3. **Páginas simples com GET ou POST** – Outra forma de criar um web service em qualquer linguagem é fazer uma página web qualquer que, ao receber uma requisição por GET ou POST, vai retornar os dados no formato XML ou JSON, em vez de retornar uma página HTML.

Essa rápida explicação visa informar os termos necessários para você fazer uma pesquisa no Google e complementar seus estudos, caso ache necessário.

Neste livro, vamos utilizar arquivos XML e JSON estáticos que coloquei no site, apenas para simular um web service. Conforme já vimos, existem três arquivos para cada tipo de carro e podemos optar por ler os dados em XML ou JSON.

```
// XML e JSON dos carros clássicos
http://www.livroandroid.com.br/livro/carros/carros_classicos.xml
http://www.livroandroid.com.br/livro/carros/carros_classicos.json

// XML e JSON dos carros esportivos
http://www.livroandroid.com.br/livro/carros/carros_esportivos.xml
http://www.livroandroid.com.br/livro/carros/carros_esportivos.json

// XML e JSON dos carros de luxo
http://www.livroandroid.com.br/livro/carros/carros_luxo.xml
http://www.livroandroid.com.br/livro/carros/carros_luxo.json
```

Para fazer uma requisição HTTP no servidor e buscar os dados no formato String, podemos utilizar a famosa classe `java.net.HttpURLConnection` do Java, mas mesmo com ela são necessárias algumas linhas de código. Para facilitar o código do projeto, criei a classe `livroandroid.lib.utils.HttpHelper` na biblioteca `android-utils`, que faz a requisição HTTP em uma única linha de código.

```
String url = "http://www.livroandroid.com.br/livro/carros/carros_classicos.json"
String json = HttpHelper.doGet(url);
```

17.2 Requisição HTTP para consultar o web service

Para continuar o desenvolvimento do projeto dos carros, vou escolher o formato de arquivo JSON, pois para o mobile essa estrutura é mais compacta que o XML e consequentemente diminui a quantidade de dados trafegados.

Para acessar a URL do web service, adicione esta constante na classe CarroService:

```
private static final String
    URL = "http://www.livroandroid.com.br/livro/carros/carros_{tipo}.json";
```

Conforme o tipo (clássicos, esportivos, luxo) solicitado, vamos substituir o texto {tipo} pelo tipo do carro, para criar a URL correta web service. No código da classe CarroService, vamos substituir o método readFile(context, tipo) que estávamos usando pelo código que faz a requisição HTTP no web service.

Veja que removi o try/catch que estava no método getCarros(context, string) e o fiz lançar a exceção IOException. Isso é o correto, pois quando a activity chamar este método para listar os carros, ela deve tratar qualquer erro.



CarroService.java

```
. . .
import livroandroid.lib.utils.HttpHelper;
public class CarroService {
    private static final String
        URL = "http://www.livroandroid.com.br/livro/carros/carros_{tipo}.json";
    public static List<Carro> getCarros(Context context, String tipo) throws IOException {
        String url = URL.replace("{tipo}", tipo);
        // Faz a requisição HTTP no servidor e retorna a string com o conteúdo.
        String json = HttpHelper.doGet(url);
        List<Carro> carros = parserJSON(context, json);
        return carros;
    }
    private static List<Carro> parserJSON(Context context, String json) {
        // Mesmo código aqui...
    }
}
```

O método CarroService.getCarros(context, string) recebe o tipo do carro desejado, que pode ser esportivos, luxo ou clássicos. Baseado no tipo do carro, a URL é montada para buscar os carros no servidor. Uma vez que temos a URL do web service, basta uma linha de código para fazer a requisição HTTP por GET e obter o resultado.

```
// Faz uma requisição do tipo GET e retorna a resposta em String
String json = HttpHelper.doGet(url);
```

A classe `HttpHelper` contém métodos para fazer a requisição HTTP por `GET` ou `POST`, e você pode utilizá-la como base para seus estudos e projetos. Internamente essa classe apenas encapsula a famosa classe `java.net.HttpURLConnection` do Java. Existem outras bibliotecas para fazer requisições HTTP no Android, como `OkHttp` e `Volley` do próprio Google. Recomendo que você as estude como complemento aos seus estudos.

Importante: no método `getCarros(context,string)`, removi o trecho de código que tinha o `try/catch` para propagar a exceção para quem chamar esse método. Isso é o correto e fará o aplicativo travar na próxima execução. Logo veremos por quê.

O restante do código da classe `CarroService` não precisamos mudar. Mas é importante que você tenha percebido que não mais usamos os arquivos da pasta `/res/raw`, pois estamos lendo o JSON diretamente do arquivo que está no site do livro. Lembrando também que no meu site esses arquivos são estáticos e apenas simulam o retorno de um web service, mas, para os objetivos didáticos deste livro, isso é suficiente.

Depois de alterar a classe `CarroService` para fazer a requisição HTTP no web service, execute o projeto novamente. O resultado é que a aplicação vai travar, mostrando o famoso **Force Close** para o usuário. Se olharmos nos logs, veremos a seguinte mensagem de erro (stack trace):

```
br.com.livroandroid.carros E/AndroidRuntime? FATAL EXCEPTION: main
Process: br.com.livroandroid.carros, PID: 988
android.os.NetworkOnMainThreadException
at android.os.StrictMode.onNetwork(StrictMode.java:1147)
```

Conforme já estudamos anteriormente no capítulo 10, sobre threads e `Handler`, esse erro acontece porque não podemos acessar a internet na `UI Thread`, portanto precisamos criar uma thread ou `AsyncTask` para desvincular o processamento da thread principal, e, se possível, ainda mostrar uma animação no estilo “por favor, aguarde” para o usuário, pois a consulta pode demorar, no caso de um dispositivo utilizando 3G.

17.3 Utilizando a classe `AsyncTask`

No capítulo 10, sobre `Handler`, expliquei o básico sobre threads, e comentei o uso da classe `AsyncTask`. Agora, vamos praticar mais uma vez.

A seguir, podemos ver o código-fonte da classe `CarrosFragment` que utiliza corretamente a classe `AsyncTask`:



CarrosFragment.java

```

...
public class CarrosFragment extends BaseFragment {
    ...
    private void taskCarros() {
        // Busca os carros: Dispara a Task
        new GetCarrosTask().execute();
    }
    // Task para buscar os carros
    private class GetCarrosTask extends AsyncTask<Void,Void,List<Carro>> {
        @Override
        protected List<Carro> doInBackground(Void... params) {
            try {
                // Busca os carros em background (Thread)
                return CarroService.getCarros(getContext(), tipo);
            } catch (IOException e) {
                Log.e("livroandroid", e.getMessage(), e);
                return null;
            }
        }
        // Atualiza a interface
        protected void onPostExecute(List<Carro> carros) {
            if(carros != null) {
                CarrosFragment.this.carros = carros;
                // Atualiza a view na UI Thread
                recyclerView.setAdapter(new CarroAdapter(getContext(), carros, onClickCarro()));
            }
        }
    }
}

```

Nesse código criamos uma `AsyncTask` para consultar o web service em segundo plano (background) e atualizar a view na UI Thread. Esta linha dispara a thread:

```
new GetCarrosTask().execute();
```

A `AsyncTask` gerencia um pool de threads internamente e vai executar o método `doInBackground()` em segundo plano, que neste caso está retornando a lista de carros. Internamente a `AsyncTask` contém um `Handler` que é utilizado para chamar o método `onPostExecute(List<Carro>)` na UI Thread. A vantagem de utilizar a `AsyncTask` é justamente essa facilidade, pois não precisamos nos preocupar com o gerenciamento das threads e a utilização de `Handlers`, conforme já estudamos.

Dica: o método `doInBackground()` da `AsyncTask` deve executar a lógica pesada que pode demorar, como consultar um web service ou banco de dados. O método `onPostExecute()` é chamado na UI Thread a fim de atualizar a interface.

Como o aplicativo vai acessar a internet, é necessário declarar a permissão no arquivo `AndroidManifest.xml`. Mas isso é só para sua revisão, pois no capítulo 14, sobre `WebView`, já fizemos essa configuração.



AndroidManifest.xml

```
<manifest ... />
    <uses-permission android:name="android.permission.INTERNET" />
    <application ... />
</manifest>
```

Ao executar o projeto novamente, a consulta no web service será feita e você deverá ver a lista de carros conforme a figura 17.1. Isso é muito legal, pois desta vez a lista de carros está online, e essa conectividade é o que dá vida à palavra “mobilidade”.

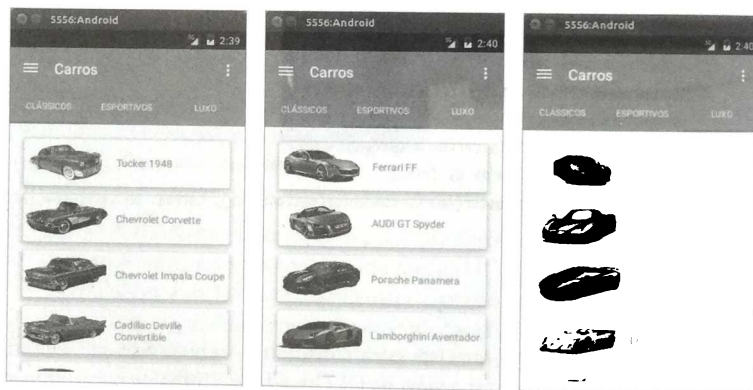


Figura 17.1 – Lista de carros do web service.

17.4 Biblioteca simples para encapsular a `AsyncTask`

No tópico anterior, buscamos os carros do web service utilizando a famosa classe `AsyncTask`, mas eu particularmente costumo utilizar uma pequena biblioteca que encapsula o acesso à `AsyncTask`.

Ao utilizar a `AsyncTask`, temos de controlar várias coisas manualmente, como por exemplo: mostrar um `ProgressBar` ou outra animação durante a execução da tarefa, tratar as possíveis exceções que são lançadas, ter uma maneira de criar várias tarefas e cancelá-las, além de tratar vários probleminhas comuns com o Android quando o dispositivo é rotacionado entre vertical e horizontal.

Na verdade, eu não quero entrar nesses assuntos avançados da `AsyncTask` neste momento do livro, pois acho cedo para termos essa conversa. O objetivo do livro é ir aumentando o grau de dificuldade durante a leitura, ao mesmo tempo em que vamos conectando diversos conceitos, e justamente por isso estamos fazendo esse projeto dos carros passo a passo.

Portanto, por ora, vamos utilizar esta pequena biblioteca que criei para disparar tarefas em segundo plano, e depois recomendo que você leia o capítulo 31, sobre `AsyncTask`, disponível no final do livro. Com o tempo, você pode estudar o código-fonte da biblioteca que vou mostrar e você vai perceber que apenas encapsulei coisas que você um dia também terá de fazer.

Bem, chega de teoria, então vamos lá!

A classe `livroandroid.lib.fragment.BaseFragment` da biblioteca `android-utils` contém o método `startTask(código,taskListener)`, que basicamente vai disparar uma `AsyncTask` e controlar a sua execução. Como parâmetro do método `startTask(código,taskListener)`, você deve informar uma implementação da interface `TaskListener`.

```
public interface TaskListener<T> {  
    // Executa em background numa Thread e retorna o objeto  
    T execute() throws Exception;  
    // Atualiza a view na UI Thread  
    void updateView(T response);  
    // Chamado caso o método execute() lance uma exceção  
    void onError(Exception exception);  
    // Chamado caso a task tenha sido cancelada  
    void onCancelled(String cod);  
}
```

Nota: a notação `<T>` é um tipo genérico e permite que esta interface seja criada com qualquer tipo, por exemplo, com `TaskListener<Carro>`, para que o `T` vire um objeto `Carro` em tempo de execução. Se achar necessário, recomendo procurar uma literatura adicional sobre Generics, pois o assunto é bem amplo e interessante.

Acredito que os métodos da interface `TaskListener` sejam autoexplicativos e até se parecem com os métodos `doInBackground()` e `onPostExecute()` da classe `AsyncTask`. Portanto, vamos atualizar o código da classe `CarrosFragment` para utilizar o método `startTask(código,taskListener)`, e durante o desenvolvimento do aplicativo vou reforçando algumas das vantagens dessa biblioteca. O código anterior que usava a `AsyncTask` diretamente você pode remover.

CarrosFragment.java

```

...
public class CarrosFragment extends BaseFragment {
    ...
    private void taskCarros() {
        // Busca os carros: Dispara a Task
        startTask("carros", new GetCarrosTask());
    }
    // Task para buscar os carros
    private class GetCarrosTask implements TaskListener<List<Carro>> {
        @Override
        public List<Carro> execute() throws Exception {
            // Busca os carros em background (Thread)
            return CarroService.getCarros(getContext(), tipo);
        }
        @Override
        public void updateView(List<Carro> carros) {
            if (carros != null) {
                // Salva a lista de carros no atributo da classe
                CarrosFragment.this.carros = carros;
                // Atualiza a view na UI Thread
                recyclerView.setAdapter(new CarroAdapter(getContext(), carros, onClickCarro()));
            }
        }
        @Override
        public void onError(Exception e) {
            // Qualquer exceção lançada no método execute vai cair aqui.
            alert("Ocorreu algum erro ao buscar os dados.");
        }
        @Override
        public void onCancelled(String s) {
        }
    }
}

```

Ao executar o projeto novamente a lista de carros será mostrada da mesma forma que antes. Mas veja que durante o download foi exibido um `ProgressDialog`, conforme a figura 17.2.



Figura 17.2 – `ProgressDialog` e lista de carros.

Esse é o comportamento padrão ao disparar as tarefas com o método `startTask(código, taskListener)` da biblioteca `android-utils`, pois um `ProgressDialog` é utilizado para informar ao usuário que a aplicação está processando alguma informação.

Mas para melhorar a experiência do usuário, vamos adicionar um `ProgressBar` no layout do fragment, exatamente por cima da lista de carros.

```

/res/layout/fragment_carros.xml

<?xml version="1.0" encoding="utf-8"?>
<FrameLayout . . .>
    <android.support.v7.widget.RecyclerView . . . />
    <ProgressBar android:id="@+id/progress"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="center" android:visibility="invisible"/>
</FrameLayout>

```

A única coisa que precisamos fazer no código é passar no terceiro parâmetro do método `startTask(código, taskListener, progress)` o identificador `R.id.progress` do `ProgressBar`.

CarrosFragment.java

```

...
public class CarrosFragment extends BaseFragment {
    private void taskCarros() {
        startTask("carros", new GetCarrosTask(), R.id.progress);
    }
}

```

O resultado dessa alteração pode ser visto na figura 17.3, que mostra a animação do `ProgressBar` durante a execução da tarefa. Lembrando que esse tipo de animação é muito importante para melhorar a experiência do usuário, principalmente no caso de o aplicativo ser acessado com uma conexão 3G muito lenta.

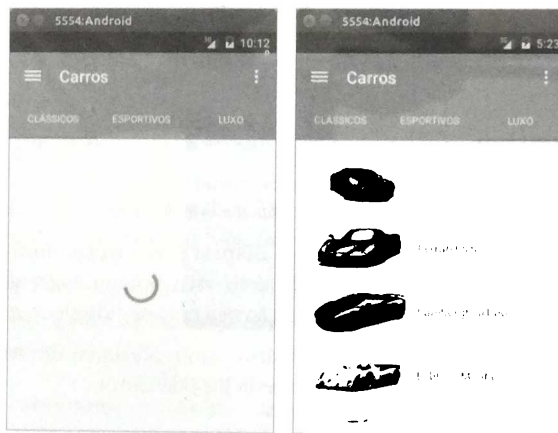


Figura 17.3 – `ProgressBar` e lista de carros.

Uma dica, ou melhor, um artifício técnico, caso queira ver as animações, é dar um pequeno “sleep” dentro do método `execute()`. Algo como 300 ou 500 milissegundos já é suficiente para você ver a animação.

```

public List<Carro> execute() throws Exception {
    Thread.sleep(500);
    return CarroService.getCarros(getContext(), tipo);
}

```

Como vimos, utilizar uma pequena classe que encapsula a `AsyncTask` tem algumas vantagens, pois ao utilizar a classe `AsyncTask` você terá de mostrar e esconder o `ProgressBar` manualmente.

Dica: a animação do `ProgressBar` é suave, recomendo utilizá-la para informar ao usuário que a aplicação está processando algo. Já o `ProgressDialog` deve ser utilizado caso o usuário precise obrigatoriamente aguardar o processamento, pois a janela do alerta é no estilo modal. Lembrando que, no caso dos tablets, como podemos ter um layout com várias views espalhadas pela tela, o recomendado é que cada fragment utilize um `ProgressBar`, pois as animações de cada um serão independentes. Neste caso, teremos várias bolinhas girando espalhadas pela tela do tablet.

17.5 Atualização por Pull to Refresh

Como a lista de carros está online, vamos adicionar o `SwipeRefreshLayout` no arquivo de layout do fragment para atualizar os dados. O `SwipeRefreshLayout` é um gerenciador de layout simples que pode conter apenas um filho, portanto faça-o envolver o `RecyclerView`.

```

 /res/layout/fragment_carros.xml

<?xml version="1.0" encoding="utf-8"?>
<FrameLayout . . . >
    <android.support.v4.widget.SwipeRefreshLayout
        android:id="@+id/swipeToRefresh"
        android:layout_width="match_parent" android:layout_height="match_parent" >
        <android.support.v7.widget.RecyclerView . . . />
    </android.support.v4.widget.SwipeRefreshLayout>
    <ProgressBar . . . />
</FrameLayout>

```

Na classe `CarrosFragment`, configure o `SwipeRefreshLayout` e adicione o método `OnRefreshListener()` para atualizar a lista.

```

 CarrosFragment.java

public class CarrosFragment extends BaseFragment {
    private SwipeRefreshLayout swipeLayout;
    . . .
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState){
        . . .
        // Swipe to Refresh
        swipeLayout = (SwipeRefreshLayout) view.findViewById(R.id.swipeToRefresh);
        swipeLayout.setOnRefreshListener(OnRefreshListener());
        swipeLayout.setColorSchemeResources(

```

```

        R.color.refresh_progress_1,
        R.color.refresh_progress_2,
        R.color.refresh_progress_3);

    return view;
}

private SwipeRefreshLayout.OnRefreshListener OnRefreshListener() {
    return new SwipeRefreshLayout.OnRefreshListener() {
        @Override
        public void onRefresh() {
            // Atualiza ao fazer o gesto Pull to Refresh
            taskCarros();
        }
    };
}

...
}

```

Como já utilizamos o `SwipeRefreshLayout` anteriormente, acredito que esse código seja tranquilo. Ao tocar na lista e arrastar o dedo para baixo, é feita a atualização dos dados, conforme a figura 174. Há, porém, um problema, pois temos duas animações na tela. Na parte superior, existe a bolinha que fica girando, referente ao `SwipeRefreshLayout`. E na parte central da tela temos o `ProgressBar`. Esse problema acontece porque a animação do `SwipeRefreshLayout` é feita automaticamente logo ao iniciar o gesto, mas, quando o método `startTask(codigo,taskListener,progress)` é chamado, o `ProgressBar` que está no centro do layout é ativado e também faz a animação.



Figura 174 – Pull to Refresh.

Para solucionar o problema, temos de mostrar uma animação, ou outra. Na verdade, o recomendado é que, na primeira vez que abrir o aplicativo e a lista estiver vazia, o `ProgressBar` seja utilizado, pois ele ficará bem sobre o fundo vazio, conforme a figura 17.5. E quando atualizarmos a lista podemos mostrar apenas a animação do `SwipeRefreshLayout`, conforme ilustrado no lado direito da figura.

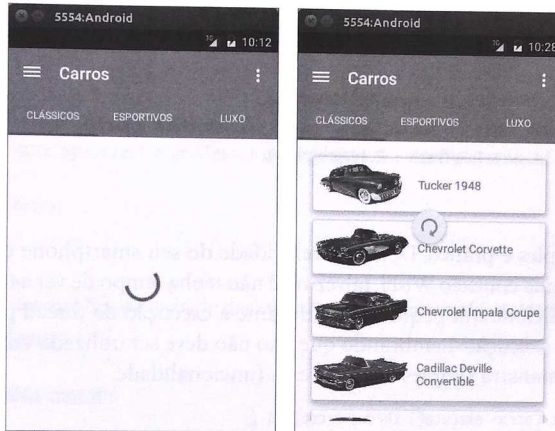


Figura 17.5 – Atualização da lista de carros.

Agora que já sabemos o que precisamos fazer, vamos colocar a mão na massa.

Apenas para revisar os conceitos, lembre-se de que o componente `SwipeRefreshLayout` inicia automaticamente a sua animação, e para parar a animação você deve chamar o método `setRefreshing(false)`. Mas a biblioteca `android-utils` já vai fazer isso por você, basta ao chamar o método `startTask(código, taskListener, progress)` informar o identificador do `ProgressBar` ou do `SwipeRefreshLayout` conforme desejado.

Para controlar quando utilizar uma animação ou outra, vamos criar um parâmetro do tipo booleano para o método `taskCarros(boolean)`, a fim de que seja possível escolher entre animar o `ProgressBar` ou o `SwipeRefreshLayout`.



CarrosFragment.java

```
public class CarrosFragment extends BaseFragment {
    ...
    private SwipeRefreshLayout.OnRefreshListener OnRefreshListener() {
        return new SwipeRefreshLayout.OnRefreshListener() {
            @Override
            public void onRefresh() {
```

```

        taskCarros(true);
    }
};
}
@Override
public void onActivityCreated(@Nullable Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);
    taskCarros(false);
}
private void taskCarros(boolean pullToRefresh) {
    startTask("carros", new GetCarrosTask(), pullToRefresh
        ? R.id.swipeToRefresh : R.id.progress);
}
}
}

```

Pronto! Simples e prático. Devido à velocidade do seu smartphone ou internet, no caso de uma conexão Wi-Fi, talvez você não tenha tempo de ver as animações. Se quiser, adicione um pequeno **sleep** durante a execução da thread para atrasar um pouco a execução. Lembrando que isso não deve ser utilizado em produção, mas é uma maneira simples de testar essa funcionalidade.

```

public List<Carro> execute() throws Exception {
    Thread.sleep(500);
    return CarroService.getCarros(getContext(), tipo);
}

```

17.6 Verificando se existe conexão disponível

Por enquanto, estamos fazendo o caminho feliz, mas na prática muitas vezes a conexão com a internet não está disponível. Então, antes de buscar os carros, precisamos verificar isso. Para testar a disponibilidade da conexão, podemos utilizar o método `AndroidUtils.isNetworkAvailable(context)` da biblioteca `android-utils`.



AndroidUtils.java

```

public class AndroidUtils {
    ...
    public static boolean isNetworkAvailable(Context context) {
        try {
            ConnectivityManager connectivity = (ConnectivityManager)
                context.getSystemService(Context.CONNECTIVITY_SERVICE);
            if (connectivity == null) {
                return false;
            }
        }
    }
}

```

```

    } else {
        NetworkInfo[] info = connectivity.getAllNetworkInfo();
        if (info != null) {
            for (int i = 0; i < info.length; i++) {
                if (info[i].getState() == NetworkInfo.State.CONNECTED) {
                    return true;
                }
            }
        }
    }
} catch (SecurityException e) {
    alertDialog(context,e.getClass().getSimpleName(), e.getMessage());
}
return false;
}
}

```

Para ler as informações da rede de dados, adicione a permissão `ACCESS_NETWORK_STATE` ao *AndroidManifest.xml*.



AndroidManifest.xml

```

<manifest ... />
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
    <application ... />
</manifest>

```

No próximo capítulo, vamos salvar todos os carros no banco de dados, e o aplicativo até poderá funcionar de forma offline. O que não pode acontecer é o usuário forçar a atualização dos dados sem existir uma conexão ativa. Vamos fazer essa verificação no método `onRefresh()`.



CarrosFragment.java

```

public class CarrosFragment extends BaseFragment {
    ...
    private SwipeRefreshLayout.OnRefreshListener OnRefreshListener() {
        return new SwipeRefreshLayout.OnRefreshListener() {
            @Override
            public void onRefresh() {
                // Valida se existe conexão ao fazer o gesto Pull to Refresh
                if(AndroidUtils.isNetworkAvailable(getContext())) {
                    taskCarros(true);
                }
            }
        };
    }
}

```

```

    } else {
        swipeLayout.setRefreshing(false);
        alert(R.string.error_conexao_indisponivel);
    }
}
};
}
}

```

Para o código compilar, adicione a seguinte mensagem ao arquivo *strings.xml*:

 /res/values/strings.xml

```

<resources>
    . . .
    <string name="error_conexao_indisponivel">Conexão indisponível. Verifique seu Wi-Fi ou 3G.</string>
</resources>

```

Depois desse tratamento, ao tentar atualizar a lista e caso a conexão não esteja disponível, o resultado será como na figura 176.



Figura 176 – Conexão indisponível.

Observe que a lógica aplicada aqui verifica se existe conexão apenas no momento de fazer o gesto de **Pull to Refresh**, porém não verifiquei se existe conexão antes de chamar a tarefa da primeira vez quando a lista está vazia. Na verdade, não fiz isso porque no próximo capítulo vamos salvar os carros no banco de dados, assim o

aplicativo poderá trabalhar de forma offline. Somente para a atualização da lista é que vamos precisar fazer alguma validação.

Outro detalhe importante é que no capítulo anterior comentei algo sobre o tratamento das exceções, e na época fiz um try/catch dentro da classe `CarroService`, o que matou a exceção, e isso não é recomendado. Mas, se você olhar o código atual da classe `CarroService`, ele não tem mais esse problema, e se alguma exceção do tipo `IOException` ou `RuntimeException` for lançada, ela será propagada, ou seja, será lançada para cima. Isso significa que, caso o método `execute()` lance uma exceção, a biblioteca `android-utils` vai tratar o erro internamente fazendo um try/catch e vai chamar o método `onError(exception)` para você fazer o tratamento adequado da exceção.

Enfim, já vimos várias vantagens de utilizar esta pequena biblioteca para disparar tarefas assíncronas.

```
private class GetCarrosTask implements TaskListener<List<Carro>> {  
    public List<Carro> execute() throws Exception {  
        // Se este método lançar uma exception...  
    }  
    public void updateView(List<Carro> carros) { }  
    public void onError(Exception e) {  
        // Vai cair aqui para você tratar o erro.  
        alert("Ocorreu algum erro ao buscar os dados: " + e.getMessage());  
    }  
    public void onCancelled(String s) { }  
}
```

17.7 Requisições HTTP com Get e Post

Conforme explicado no início deste capítulo, existem vários tipos de web services, e cada um vai exigir um tipo de comunicação. O restante deste capítulo tem como objetivo fornecer o básico sobre requisições HTTP do tipo **GET** e **POST** e também como acessar web services que são escritos com WSDL.

No aplicativo dos carros, fizemos uma requisição do tipo **GET** no servidor, para buscar o arquivo XML ou JSON. A requisição do tipo **GET** passa todos os parâmetros na URL e é utilizada frequentemente para consulta. Uma URL com **GET** tem a seguinte sintaxe:

`http://www.site.com.br?param1=valor1¶m2=valor2`

Apenas para você traçar um paralelo, sempre que você digita uma URL no browser para abrir o site desejado, é feita uma requisição do tipo **GET**. A diferença é que o

site retorna os dados no formato HTML, já o web service geralmente retorna os dados em XML ou JSON.

Por sua vez, uma requisição do tipo **POST** envia os parâmetros no corpo da requisição HTTP, e a URL apenas contém o endereço final (endpoint) em que você deseja enviar/postar determinado conteúdo.

<http://www.site.com.br>

No corpo da requisição, são enviados os parâmetros, por exemplo: {param1=valor1¶m2=valor2}. A vantagem da requisição do tipo **POST** é que, caso você precise enviar informações secretas como uma senha, ela é enviada no corpo da requisição. Se fosse utilizado o **GET**, a senha seria enviada na URL e ficaria visível junto com os demais parâmetros.

Traçando um paralelo mais uma vez, quando você faz login no Gmail ou em qualquer formulário de cadastro na internet, geralmente é feita uma requisição do tipo **POST** no site. A requisição do tipo **GET** também tem um limite em relação à quantidade de parâmetros que podem ser enviados, portanto o **POST** também é recomendado para trafegar muitas informações.

Outro exemplo em que o **POST** deve ser utilizado é para trafegar arquivos, tais como fazer o upload de fotos. Estou explicando isso porque o web service dos carros foi utilizado apenas para consulta, e isso é um clássico exemplo de utilização do **GET**. Mas se fosse necessário salvar um carro no servidor, provavelmente os dados seriam enviados por **POST**.

Eu não tenho um web service desse tipo disponível, então, para brincarmos um pouco com requisições HTTP do tipo **POST**, vamos utilizar este web service disponibilizado no site da **W3Schools**:

<http://www.w3schools.com/webservices/tempconvert.asmx>

Pela extensão *.asmx*, sabemos que esse web service foi desenvolvido em .NET, e na listagem dos serviços (Figura 17.7) podemos verificar os métodos CelsiusToFahrenheit e FahrenheitToCelsius que podemos consultar.

Ao clicar no serviço CelsiusToFahrenheit, a página vai abrir o formulário para você testar o web service por **POST** (Figura 17.8). Podemos ver que, para chamar o web service, existe apenas o parâmetro Celsius, que recebe o valor para converter de graus Celsius para Fahrenheit.

Ao preencher o campo Celsius com o valor 1 e clicar no botão Invoke, é feito o **POST** no web service. Podemos conferir o resultado na figura 17.9, que mostra o valor convertido para Fahrenheit.

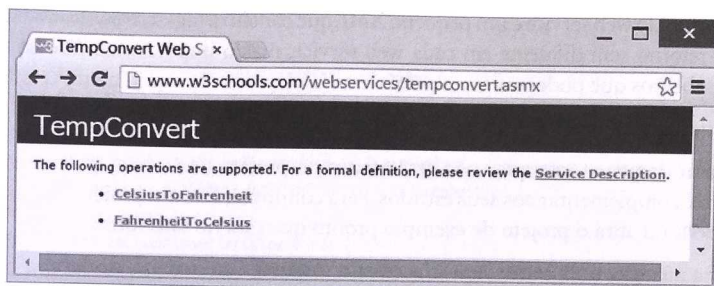


Figura 17.7 – Web service no site da W3Schools.

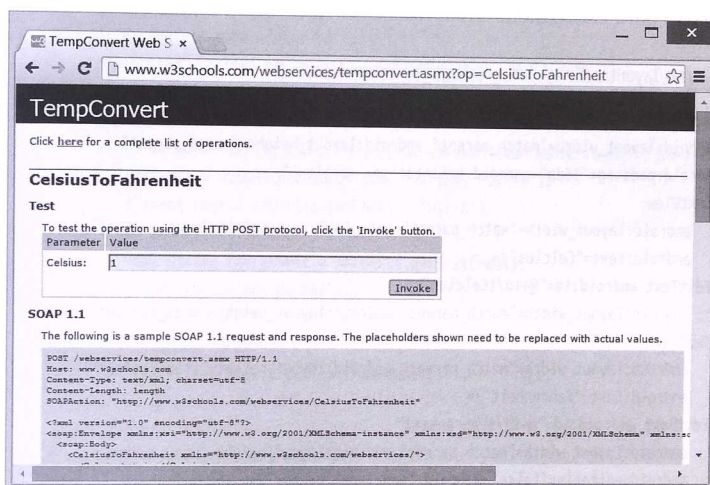


Figura 17.8 – Exemplo de web service.

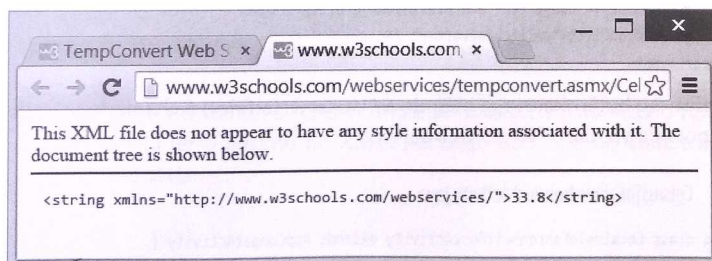


Figura 17.9 – Resultado do web service.

O retorno do web service é um pequeno XML que contém a tag `<string>valor</string>`. Esse retorno será diferente em cada web service, mas o importante nesse caso é que sabemos que podemos fazer o **POST** nesta URL passando o parâmetro Celsius,

<http://www.w3schools.com/webservices/tempconvert.aspx/CelsiusToFahrenheit>

A partir daqui, os exemplos não serão feitos no projeto dos carros, pois esse assunto é complementar aos seus estudos. Para continuar, crie um projeto chamado HelloWorld ou abra o projeto de exemplo pronto que está no GitHub.

Tenha atenção, pois como vamos acessar a internet é necessário declarar a permissão **INTERNET**. Todos os próximos exemplos vão utilizar o mesmo arquivo de layout, que é um simples formulário que permite digitar o valor em Celsius para converter para Fahrenheit.

 `/res/layout/activity_form_celsius_to_fahrenheit.xml`

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="16dp" android:orientation="vertical">
    <TextView
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="Celsius"/>
    <EditText android:id="@+id/tCelsius"
        android:layout_width="match_parent" android:layout_height="wrap_content" />
    <TextView
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="Fahrenheit"/>
    <EditText android:id="@+id/tFahrenheit"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:editable="false" android:inputType="number"/>
    <Button
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Converter" android:layout_gravity="right"
        android:onClick="onClickConverter" />
</LinearLayout>
```

Ao clicar no botão **Converter**, o método `onClickConverter(view)` é chamado na activity, que por sua vez faz o **POST** no web service.

 `CelsiusToFahrenheitPostActivity.java`

```
public class CelsiusToFahrenheitPostActivity extends AppCompatActivity {
    String URL = "http://www.w3schools.com/webservices/tempconvert.aspx/CelsiusToFahrenheit";
    private EditText tCelsius;
```

```

private EditText tFahrenheit;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_form_celsius_to_fahrenheit);
    tCelsius = (EditText) findViewById(R.id.tCelsius);
    tFahrenheit = (EditText) findViewById(R.id.tFahrenheit);
}

public void onClickConverter(View view) {
    new Thread() {
        @Override
        public void run() {
            String celsius = tCelsius.getText().toString();
            // Parâmetros para enviar por post
            Map<String, String> params = new HashMap<String, String>();
            params.put("Celsius", celsius);
            try {
                // Retorno: <string xmlns="http://www.w3schools.com/webservices/">33.8</string>
                String s = HttpHelper.doPost(URL, params, "UTF-8");
                Element root = XMLUtils.getRoot(s, "UTF-8");
                // Lê o texto do XML
                final String fahrenheit = XMLUtils.getText(root);
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        tFahrenheit.setText(fahrenheit);
                    }
                });
            } catch (IOException e) {
                Log.e("livroandroid", "Erro: " + e.getMessage(), e);
            }
        }
    }.start();
}
}

```

Observe que estou utilizando as classes `HttpHelper` e `XMLUtils` para fazer o **POST** no web service e para ler o retorno do XML, portanto declare a dependência da biblioteca `android-utils`.

 app/build.gradle

```
dependencies { . . .  
    compile 'br.com.livroandroid:android-utils:1.0.0'  
}
```

A figura 17.10 mostra o resultado da conversão de Celsius para Fahrenheit.

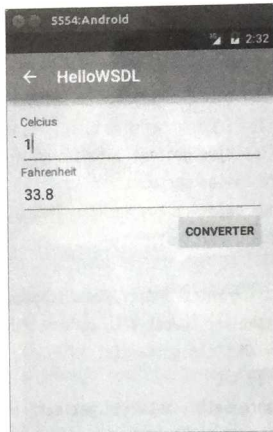


Figura 17.10 – Exemplo de Post.

17.8 Web services com WSDL

No aplicativo dos carros, estamos utilizando os arquivos XML e JSON que estão no site para simular um web service, pois esses dois tipos de retornos são leves e essa arquitetura é bem simples e muito utilizada.

Mas, dependendo do serviço que existe no servidor, às vezes é necessário consumir web services clássicos, que são descritos por uma WSDL (Web Service Definition Language). Neste caso, uma mensagem SOAP (Simple Object Access protocol) é enviada para trafegar os dados. Enfim, este não é um livro de web services, então, se for necessário, procure alguma literatura adicional.

O fato é que às vezes precisamos acessar esses tipos de serviços, e para isso precisamos gerar um cliente de web service, a fim de chamar os métodos remotos que foram definidos no servidor. No Android, podemos utilizar a biblioteca *KSoap* para acessar esses web services. Essa biblioteca é pequena e leve, e basicamente encapsula o trabalho chato de criar as mensagens SOAP.

<http://kobjects.org/ksoap2/>

<https://code.google.com/p/ksoap2-android/>

Como estamos utilizando o Android Studio e o Gradle, não precisamos nos preocupar em baixar essa biblioteca; basta adicionar uma linha com a dependência no arquivo *app/build.gradle*, conforme demonstrado a seguir.



app/build.gradle

```
dependencies {
    . . .
    compile 'br.com.livroandroid:android-utils:1.0.0'
    // Biblioteca KSOAP2 para web services WSDL
    compile 'com.google.code.ksoap2-android:ksoap2-android:3.1.1'
}
```

Para testar o web service, vamos utilizar o mesmo serviço disponibilizado no site da W3Schools.

<http://www.w3schools.com/webservices/tempconvert.asmx>

Nesta página, se você clicar no link **Service Description**, a página vai redirecionar para o seguinte link, que apenas adiciona o parâmetro ?WSDL no final da URL:

<http://www.w3schools.com/webservices/tempconvert.asmx?WSDL>

Ao fazer isso, você verá o arquivo WSDL desse serviço, que basicamente declara todos os métodos que o web service contém. O que geralmente fazemos é gerar um cliente de web service, utilizando ferramentas que geram código a partir desse arquivo WSDL. Mas a biblioteca KSoap oferece uma alternativa para esta geração de código e permite você chamar os métodos desse web service manualmente. Para chamar um método de um web service, precisamos montar um XML no formato do protocolo SOAP e fazer post na página do web service. Como isso é trabalhoso, o KSoap permite que você apenas configure o nome do método, os parâmetros que deseja enviar e internamente ele faz o trabalho pesado para criar a mensagem no formato SOAP.

A seguir, podemos verificar um exemplo de código que mostra como executar o método *CelsiusToFahrenheit* deste web service.



CelsiusToFahrenheitKSoapActivity.java

```
public class CelsiusToFahrenheitKSoapActivity extends AppCompatActivity {
    . . .
    public void onClickConverter(View view) {
```

```

        . . .
        String celcius = tCelcius.getText().toString();
        final String fahrenheit = CelsiusToFahrenheit(URL, celcius);
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                tFahrenheit.setText(fahrenheit);
            }
        });
    }

    // Faz o POST com KSOAP e retorna o resultado.
    public String CelsiusToFahrenheit(String url, String celcius) throws Exception {
        SoapSerializationEnvelope soapEnvelope = new
            SoapSerializationEnvelope(SoapEnvelope.VER11);
        soapEnvelope.implicitTypes = true;
        soapEnvelope.dotNet = true;
        SoapObject soapReq = new
            SoapObject("http://www.w3schools.com/webservices/", "CelsiusToFahrenheit");
        soapReq.addProperty("Celcius", celcius);
        soapEnvelope.setOutputSoapObject(soapReq);
        int timeout = 60000;
        HttpTransportSE httpTransport = new HttpTransportSE(url, timeout);
        try {
            // Faz a chamada
            httpTransport.call("http://www.w3schools.com/webservices/CelsiusToFahrenheit",
                soapEnvelope);
            // Lê o retorno
            Object retObj = soapEnvelope.bodyIn;
            if (retObj instanceof SoapFault) {
                SoapFault fault = (SoapFault) retObj;
                Exception ex = new Exception(fault.faultstring);
                throw ex;
            } else {
                // Retorno OK
                SoapObject result = (SoapObject) retObj;
                if (result.getPropertyCount() > 0) {
                    Object obj = result.getProperty(0);
                    if (obj != null && obj.getClass().equals(SoapPrimitive.class)) {
                        SoapPrimitive j = (SoapPrimitive) obj;
                        String resultVariable = j.toString();
                        return resultVariable;
                    }
                }
            }
        }
    }

```

```
        } else if (obj != null && obj instanceof String) {  
            String resultVariable = (String) obj;  
            return resultVariable;  
        }  
    }  
}  
}  
} catch (Exception e) {  
    throw e;  
}  
return "";  
}  
}
```

Nota: vale lembrar que web services com WSDL estão sendo cada vez menos utilizados como serviços, principalmente no mundo mobile. Um dos motivos disso é porque o protocolo SOAP é pesado, pois na prática ele é um grande arquivo XML que fica indo para lá e para cá, trafegando na rede. Portanto, preferencialmente, se puder escolher, crie serviços simples que retornam os dados em JSON.

Ao executar esse exemplo, o resultado será o mesmo do exemplo anterior. Outra forma de consultar os serviços de web service WSDL é gerar o código cliente, como eu disse anteriormente. No caso do Android, eu já utilizei comercialmente esta página para gerar o código cliente para chamar um web service que tinha muitos métodos:

<http://www.wsdl2code.com/>

Nessa página, basta você digitar a URL da WSDL do web service e clicar para gerar o código. É necessário fazer um cadastro antes, mas até o momento que este livro estava sendo escrito esse serviço era gratuito. Depois de gerar o código cliente do serviço, copie as classes para o projeto do Android Studio. No caso deste serviço da **W3Schools**, a classe que foi gerada chama-se `TempConvert` e basicamente encapsula a API do `KSoap2`, facilitando a vida do desenvolvedor para utilizar o web service. Internamente, essa classe vai utilizar exatamente a mesma API do `KSoap2` que já estudamos.

A seguir, temos um exemplo de código que utiliza a classe `TempConvert` que foi gerada automaticamente pela página *www.wsdl2code.com*. Veja como o código final é simples. Podemos concluir que, se for necessário acessar web services criados com WSDL, vale a pena utilizar algum gerador de código como fizemos aqui.



CelsiusToFahrenheitActivity.java

```
public class CelsiusToFahrenheitActivity extends AppCompatActivity {
    ...
    public void onClickConverter(View view) {
        new Thread() {
            public void run() {
                TempConvert t = new TempConvert();
                String celcius = tCelcius.getText().toString();
                final String fahrenheit = t.CelsiusToFahrenheit(celcius);
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        tFahrenheit.setText(fahrenheit);
                    }
                });
            }
        }.start();
    }
}
```

17.9 Links úteis

Neste capítulo, aprendemos a consultar web services e ler o retorno nos formatos XML e JSON, além de termos estudado como consultar serviços com WSDL.

Lembre-se de que, sempre que acessar a internet, é preciso abrir uma thread para não travar o processamento, e para isso utilizamos o método `startTask(codigo, taskListener, progress)` da biblioteca `android-utils`. Eu particularmente faço assim em meus projetos, e inclusive utilizo uma versão melhorada e com mais métodos da mesma classe `HttpHelper` que mostrei aqui.

Existem muitas outras bibliotecas para consultar web services no Android, e devido à variedade cada desenvolvedor acaba optando por utilizar a biblioteca que mais o agrada.

Apenas para fechar o capítulo, vale a pena explicar um clássico bug referente a requisições HTTP no Android. Desde o início no Android, existem duas classes que podem fazer requisições HTTP: a classe `URLConnection` e a classe `HttpClient` da famosa biblioteca da Apache. Mas, segundo um post oficial no blog do Google Developers, a biblioteca `URLConnection` apresenta alguns bugs até a versão 2.2 do

Android, e deve-se utilizar a biblioteca `HttpClient`. No entanto, do Android 2.3 ou superior, esse bug foi resolvido, e o Google recomenda utilizar a classe `URLConnection`. Para evitar esta salada de fruta, podemos utilizar uma biblioteca que encapsula esses problemas e faz a correta utilização destas APIs dependendo da versão do Android.

Esta biblioteca é a `OkHttp`, que está disponível no seguinte endereço:

<http://square.github.io/okhttp/>

Sua utilização é bem simples, e recomendo que você dê uma olhada. Eu não posso demonstrar exemplos de cada uma das bibliotecas neste livro, mas recomendo uma leitura complementar para complementar seus estudos.

- **Android Developers Blog** – Explicação do problema clássico com `URLConnection` vs `HttpClient`.

<http://android-developers.blogspot.com.br/2011/09/androids-http-clients.html>

- **OkHttp** – Biblioteca para fazer requisições HTTP no Android.

<http://square.github.io/okhttp/>

- **Retrofit** – Biblioteca muito famosa para acessar serviços REST.

<http://square.github.io/retrofit/>

- **Volley** – Biblioteca oficial do Google para fazer requisições HTTP de forma assíncrona. Recomendo assistir a palestra feita no Google I/O 2013 sobre o Volley.

<http://developer.android.com/training/volley>

E para download de imagens existem também algumas bibliotecas bem populares na comunidade de desenvolvedores Android.

- **Picasso** – Já conhecemos essa.

<http://square.github.io/picasso/>

- **Universal Image Loader** – Outra biblioteca para fazer download de imagens.

<https://github.com/nostra13/Android-Universal-Image-Loader>

- **Android-Query (AQuery)** – Outra biblioteca para fazer download de imagens.

<https://code.google.com/p/android-query/>



CAPÍTULO 18

Persistência

O Android tem integração com o SQLite, um leve e poderoso banco de dados, permitindo que você utilize banco de dados normalmente em sua aplicação.

Embora o armazenamento em banco de dados seja a forma mais comum de persistência, o Android também permite que arquivos sejam salvos na memória interna ou no SD card e também tem um sistema simples de persistência de chave e valor chamado de preferências.

18.1 Salvando as preferências do usuário com a classe `SharedPreferences`

Para começar a brincadeira, vamos aprender a salvar dados simples no formato de chave e valor como se fosse uma `HashTable`. Para isso, podemos usar a classe `android.content.SharedPreferences`, que salva automaticamente os dados em um banco de dados interno da aplicação.

A classe `SharedPreferences` deve ser utilizada para salvar valores pequenos, como tipos primitivos e pequenas strings. Para encapsular as chamadas para a classe `SharedPreferences`, costumo criar uma classe chamada `Prefs`, lembrando o termo “preferências”, pois frequentemente essa funcionalidade é chamada de salvar uma informação nas preferências do usuário.

Essa classe faz parte da biblioteca `android-utils`, então o código-fonte a seguir é apenas para sua referência. Os métodos `setBoolean(context,chave,valor)`, `setInteger(context,chave,valor)` e `setString(context,chave,valor)` salvam os valores no banco de dados interno do Android. e os métodos `getBoolean(context,chave)`, `getInteger(context,chave)` e `getString(context,chave)` leem os valores salvos do banco de dados.



`Prefs.java`

```
package livroandroid.lib.utils;  
import android.content.Context;
```

```
import android.content.SharedPreferences;

public class Prefs {
    // Identificador do banco de dados destas preferências
    public static final String PREF_ID = "livroandroid";
    public static void setBoolean(Context context, String chave, boolean on) {
        SharedPreferences pref = context.getSharedPreferences(PREF_ID, 0);
        SharedPreferences.Editor editor = pref.edit();
        editor.putBoolean(chave, on);
        editor.commit();
    }

    public static boolean getBoolean(Context context, String chave) {
        SharedPreferences pref = context.getSharedPreferences(PREF_ID, 0);
        boolean b = pref.getBoolean(chave, true);
        return b;
    }

    public static void setInteger(Context context, String chave, int valor) {
        SharedPreferences pref = context.getSharedPreferences(PREF_ID, 0);
        SharedPreferences.Editor editor = pref.edit();
        editor.putInt(chave, valor);
        editor.commit();
    }

    public static int getInteger(Context context, String chave) {
        SharedPreferences pref = context.getSharedPreferences(PREF_ID, 0);
        int i = pref.getInt(chave, 0);
        return i;
    }

    public static void setString(Context context, String chave, String valor) {
        SharedPreferences pref = context.getSharedPreferences(PREF_ID, 0);
        SharedPreferences.Editor editor = pref.edit();
        editor.putString(chave, valor);
        editor.commit();
    }

    public static String getString(Context context, String chave) {
        SharedPreferences pref = context.getSharedPreferences(PREF_ID, 0);
        String s = pref.getString(chave, "");
        return s;
    }
}
```

Nota: ao salvar um valor com a classe `SharedPreferences`, é criado um objeto do tipo `android.content.SharedPreferences.Editor` e depois é chamado o método `commit()`, que efetiva as alterações no banco de dados interno da aplicação.

Para praticarmos o conceito, vamos aprimorar o projeto dos carros e salvar o índice da última tab selecionada pelo usuário, assim quando o aplicativo for aberto novamente a última tab será mostrada automaticamente. No método `onTabSelected(tab)`, vamos salvar o índice da tab selecionada, para depois recuperar o índice salvo quando o fragment for criado.

CarrosTabFragment.java

```
...
import livroandroid.lib.utils.Prefs;
public class CarrosTabFragment extends BaseFragment {
    private TabLayout tabLayout;
    private ViewPager viewPager;
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        ...
        // Cria o ViewPager e as tabs...

        // Ao criar a view, mostra a última tab selecionada
        int tabIdx = Prefs.getInteger(getContext(), "tabIdx");
        viewPager.setCurrentItem(tabIdx);
        return view;
    }
    @Override
    public void onTabSelected(TabLayout.Tab tab) {
        // Se alterar a tab, atualiza o ViewPager
        mViewPager.setCurrentItem(tab.getPosition());
        // Salva o índice da página/tab selecionada
        Prefs.setInteger(getContext(), "tabIdx", mViewPager.getCurrentItem());
    }
}
```

Feitas essas alterações, execute o projeto novamente e selecione alguma tab. Depois feche o aplicativo com o botão voltar e inicie o aplicativo novamente. O resultado é que a aplicação será iniciada com a última tab/página selecionada.

Neste exemplo, utilizamos a classe `Prefs`, que encapsula o acesso à classe `SharedPreferences`. Conforme acabamos de ver, salvar pequenas informações no formato de chave e valor é muito útil em casos simples como esse, e a vantagem é que internamente o Android cria um banco de dados interno da aplicação para armazenar essas informações.

18.2 Activity de configurações

É comum encontrar nos aplicativos telas de configurações (settings) para o usuário personalizar os dados. Sendo assim, para praticarmos, vamos implementar uma dessas telas no projeto dos carros, por isso deixamos preparado o menu **Configurações** no menu lateral.

Uma maneira de implementar esse tipo de funcionalidade é criar o layout de uma tela de configurações manualmente e utilizar a classe `SharedPreferences` para salvar os dados. Felizmente, o Android já tem um processo automatizado para isso, e a vantagem é que os dados são salvos automaticamente, além de deixar a interface de usuário coerente com o resto da plataforma.

Nosso objetivo é criar duas activities: uma compatível com o Android 2.x e outra compatível com Android 3.x ou superior. Portanto, ao selecionar o item **Configurações** no menu lateral, será aberta uma dessas duas activities, conforme mostra a figura 18.1. O lado esquerdo da figura mostra a activity executando no Android 5.0 e o direito mostra o Android 2.3. Naturalmente, cada plataforma tem o seu padrão de interface. Você não consegue ver, porque o livro é impresso em preto e branco, mas no Android 5.0, a action bar será azul (cor primária) e alguns componentes como o checkbox ficam vermelhos (cor de acentuação), pois configuramos o tema Material com essas cores.

Na tela inicial que tem o Navigation Drawer, estamos atualizando o conteúdo central do aplicativo apenas utilizando fragments. Foi dessa forma que fizemos a lista de carros e a página com o **WebView**. Mas, ao selecionar o item **Configurações** no menu, vamos abrir uma nova activity devido a problemas técnicos.

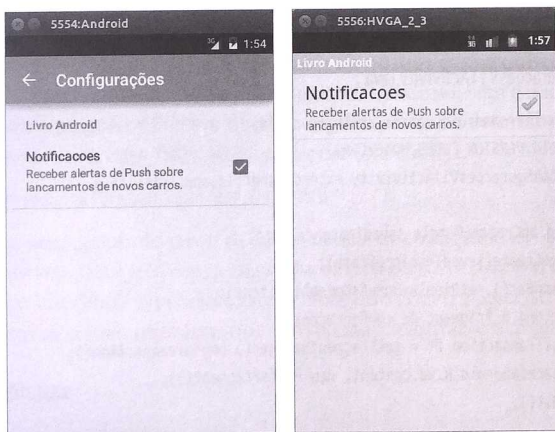


Figura 18.1 – Activity de configurações.

Até o Android 2.3, a maneira de criar esse tipo de tela era utilizando uma activity filha `android.preference.PreferenceActivity`, fato que nos obriga a abrir uma nova activity. E a partir do Android 3.0 foi criado o `fragment android.preference.PreferenceFragment`, que facilita criar telas de configurações, mas infelizmente, até o momento em que este livro está sendo escrito, esse fragment somente existe com a versão nativa (`android.app.Fragment`) e não com a biblioteca de compatibilidade (`android.support.v4.app.Fragment`). Portanto, vamos ter de criar uma activity normal (sem ser de compatibilidade) e configurá-la para utilizar o tema Material.

Dadas as devidas explicações, vamos colocar a mão na massa, então crie estas duas activities:



ConfiguracoesActivity.java

```
package br.com.livroandroid.carros.activity.prefs;
@SuppressWarnings("deprecation")
public class ConfiguracoesActivity extends android.preference.PreferenceActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Carrega as configurações
        addPreferencesFromResource(R.xml.preferences);
    }
}
```

A próxima activity chamei de V11, para lembrar que ela é compatível com Android 3.0 (API Level 11) ou superior. Note que ela declara um fragment interno do tipo `android.preference.PreferenceFragment` e o insere como a raiz do layout.



ConfiguracoesV11Activity.java

```
package br.com.livroandroid.carros.activity.prefs;
@TargetApi(Build.VERSION_CODES.HONEYCOMB)
public class ConfiguracoesV11Activity extends android.app.Activity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getActionBar().setDisplayHomeAsUpEnabled(true);
        // Adiciona o fragment de configurações
        FragmentTransaction ft = getFragmentManager().beginTransaction();
        ft.replace(android.R.id.content, new PrefsFragment());
        ft.commit();
    }
}
```

```

public static class PrefsFragment extends android.preference.PreferenceFragment {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Carrega as configurações
        addPreferencesFromResource(R.xml.preferences);
    }
}

```

O método `addPreferencesFromResource(prefsResId)` carrega um arquivo XML que contém as configurações das preferências; portanto, crie o seguinte arquivo de configuração na pasta `/res/xml`:

```

❏ /res/xml/preferences.xml

<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
    <PreferenceCategory android:title="Livro Android">
        <CheckBoxPreference
            android:key="PREF_CHECK_PUSH"
            android:summary="Receber alertas de Push sobre lançamentos de novos carros."
            android:title="Notificacoes" />
    </PreferenceCategory>
</PreferenceScreen>

```

Esse arquivo XML declara uma tela de preferências `PreferenceScreen` e cria uma categoria `PreferenceCategory` chamada **Livro Android**, sendo que uma categoria pode conter vários componentes. O item `CheckBoxPreference` mostra um checkbox para o usuário selecionar uma opção. A vantagem de utilizar esse arquivo XML é que todos os valores são salvos automaticamente pelo Android, sem a necessidade de nenhuma programação. Observe que no arquivo XML foi definido que a chave do `CheckBoxPreference` é `PREF_CHECK_PUSH`, conforme este código:

```
<CheckBoxPreference android:key="PREF_CHECK_PUSH" ... />
```

Isso significa que, depois de salvar os dados na tela de configuração, basta lermos o valor da chave `PREF_CHECK_PUSH` com a classe `SharedPreferences`. A classe `PrefsUtils` mostra como ler se o checkbox está selecionado. Note que criei o pacote util no projeto para adicionar as classes utilitárias que vamos criando durante o desenvolvimento.

```
❏ PrefsUtils.java
```

```

package br.com.livroandroid.carros.util;
public class PrefsUtils {

```



```
// Verifica se o usuário marcou o checkbox de Push ON nas configurações
public static boolean isCheckPushOn(final Context context) {
    SharedPreferences sp = PreferenceManager.getDefaultSharedPreferences(context);
    return sp.getBoolean("PREF_CHECK_PUSH", false);
}
}
```

Para a activity compatível com Android 3.0 ou superior, vamos customizar as cores do tema Material, mas não podemos utilizar o tema AppCompat, pois essa é uma activity nativa que herda de `android.app.Activity`. Sendo assim, vamos criar mais um tema chamado `AppTheme.Material`. Repare que, embora a activity seja compatível com Android 3.0 (API Level 11), vamos customizar as cores apenas para o Android 5.0 (API Level 21), pois foi quando o tema Material foi criado. Para versões anteriores, a tela de configurações terá a interface padrão de cada plataforma.

```
 /res/values/styles-v21.xml

<resources>
    <style name="AppTheme.NavDrawer" parent="AppTheme">
        . . .
    </style>
    <!-- Tema Material para a tela de configurações -->
    <style name="AppTheme.Material" parent="android:Theme.Material.Light.DarkActionBar">
        <!-- Cores -->
        <item name="android:colorPrimary">@color/primary</item>
        <item name="android:colorPrimaryDark">@color/primary_dark</item>
        <item name="android:colorAccent">@color/accent</item>
    </style>
</resources>
```

Para finalizar as configurações, declare ambas as activities no *AndroidManifest.xml*. Observe veja que a activity V11 utiliza o tema `AppTheme.Material` que acabamos de configurar. Repare também que adicionei ambas as activities no pacote `.activity.prefs`.

```
 AndroidManifest.xml

. . .
<activity android:name=".activity.prefs.ConfiguracoesActivity"
    android:label="Configurações">
    <meta-data android:name="android.support.PARENT_ACTIVITY"
        android:value=".activity.MainActivity" />
</activity>
<activity android:name=".activity.prefs.ConfiguracoesV11Activity"
    android:label="Configurações">
```

```

    android:parentActivityName=".activity.MainActivity"
    android:theme="@style/AppTheme.Material">
    <meta-data android:name="android.support.PARENT_ACTIVITY"
        android:value=".activity.MainActivity" />
</activity>

```

Nota: lembre-se de que o atributo `android:parentActivityName` somente existe para o Android 4.0 ou superior e ele define a activity pai na hierarquia para voltar a navegação caso o usuário pressione o botão de up navigation. Por questões de compatibilidade com versões anteriores, a tag `<meta-data>` é configurada com o mesmo propósito.

Uma vez que as activities estão devidamente configuradas, altere o método que trata o evento de clique do menu lateral para chamar a activity apropriada conforme a versão do Android.



MainActivity.java

```

. . .
@Override
public void onNavDrawerItemSelected(. . .) {
    if (position == 0) { . . . }
    else if (position == 1) { . . . }
    else if (position == 2) { // Abrir as configurações...
        if (AndroidUtils.isAndroid3Honeycomb()) {
            startActivity(new Intent(this, ConfiguracoesV11Activity.class));
        } else {
            startActivity(new Intent(this, ConfiguracoesActivity.class));
        }
    }
}
}

```

O resultado deve ser como a figura 18.1, que foi apresentada no início deste tópico. Lembre-se de que, depois de salvar os dados na tela de configurações, você precisa ler o valor que foi salvo, e para isso criamos a classe `PrefsUtils`.

```
boolean b = PrefsUtils.isCheckPushOn(this); // true se marcou
```

Onde inserir esse código para ler essa informação deixarei como lição de casa para você, pois o objetivo foi apenas mostrar como criar esse tipo de tela de configurações (preferências). Para complementar seus estudos, recomendo uma leitura da documentação oficial, assim você pode ver outros exemplos e mais componentes que podem ser utilizados para criar este tipo de tela.

<http://developer.android.com/guide/topics/ui/settings.html>

18.3 Lendo e salvando arquivos

Para ler e salvar arquivos no Android, são utilizadas as tradicionais interfaces `java.io.InputStream` e `java.io.OutputStream` da linguagem Java. Portanto, se você já trabalhou com arquivos em Java, vai se sentir em casa. A única coisa que precisamos fazer é obter uma pasta para salvar ou ler os arquivos, representada por um objeto do tipo `java.io.File`.

Podemos salvar arquivos na memória interna ou externa do dispositivo. A memória interna é utilizada para salvar arquivos que ficam privados para sua aplicação. A memória externa, ou seja, o cartão de memória (SD card), permite salvar os arquivos de forma pública, mas nesse caso tanto o usuário quanto outras aplicações poderão ter acesso aos arquivos.

18.4 Trabalhando com arquivos na memória interna

Para abrir arquivos privados da aplicação, podemos utilizar os seguintes métodos da classe `android.content.Context`, que só para lembrar é a classe mãe da `android.app.Activity`.

- `FileInputStream openFileInput(nome)` – Abre um arquivo para leitura, retornando um `FileInputStream`, ou a exceção `FileNotFoundException` é lançada caso o arquivo não seja encontrado.
- `FileOutputStream openFileOutput(name, modo)` – Abre um arquivo para escrita, retornando um `FileOutputStream`. No parâmetro `modo`, podem ser informadas as constantes `Context.MODE_PRIVATE`, que é o padrão, ou `Context.MODE_APPEND`, para sempre usar o arquivo já existente e ir concatenando as novas informações no final do arquivo.
- `boolean deleteFile(String nome)` – Exclui o arquivo e retorna verdadeiro se a operação foi realizada com sucesso.

Utilizando esses métodos para ler e salvar arquivos, o arquivo será criado no diretório `/data/data/pacote.do.aplicativo/files/`, o qual é privado da aplicação e justamente por isso é conhecido como memória interna. Caso o objetivo seja salvar o arquivo, podemos utilizar o método `openFileOutput("arquivo.txt", Context.MODE_PRIVATE)` para abrir o arquivo e automaticamente retornar uma `FileOutputStream` para escrever os dados. Mas caso o seu objetivo seja somente obter um objeto `java.io.File` que represente a localização do arquivo, utilize o método `getFilePath(arquivo)` da classe `Context`. Como exemplo, veja este trecho de código:

```
File file = context.getFilePath("arquivo.txt");
```

Nesse caso, é retornada a localização do arquivo (sem criá-lo):

```
/data/data/pacote.do.aplicativo/files/arquivo.txt
```

Para demonstrar como salvar arquivos na memória interna da aplicação, vamos salvar em arquivo o próprio arquivo JSON que estamos buscando do site do livro. Com esse propósito, vamos criar o método `salvaArquivoNaMemoriaInterna()` na classe `CarroService`.



CarroService.java

```
...
public class CarroService {
    ...
    public static List<Carro> getCarros(Context context, String tipo) throws IOException {
        String url = URL.replace("{tipo}", tipo);
        String json = HttpHelper.doGet(url);
        salvaArquivoNaMemoriaInterna(context, url, json);
        List<Carro> carros = parserJSON(context, json);
        return carros;
    }
    private static void salvaArquivoNaMemoriaInterna(Context context, String url, String json) {
        String fileName = url.substring(url.lastIndexOf("/") + 1);
        File file = FileUtils.getFile(context, fileName);
        IOUtils.writeString(file, json);
        Log.d(TAG, "Arquivo salvo: " + file);
    }
}
```

Feito isso, execute o projeto novamente e você verá as seguintes mensagens no LogCat, indicando que os arquivos foram salvos com sucesso.

```
Arquivo salvo: /data/data/br.com.livroandroid.carros/files/carros_classicos.json
Arquivo salvo: /data/data/br.com.livroandroid.carros/files/carros_esportivos.json
Arquivo salvo: /data/data/br.com.livroandroid.carros/files/carros_luxo.json
```

Lembre-se de que a pasta `/data/data/br.com.livroandroid.carros/files` é privada da aplicação, e para visualizarmos o seu conteúdo no emulador podemos utilizar a ferramenta `adb` disponível no Android SDK. Para visualizar os arquivos, abra um terminal e digite os seguintes comandos:

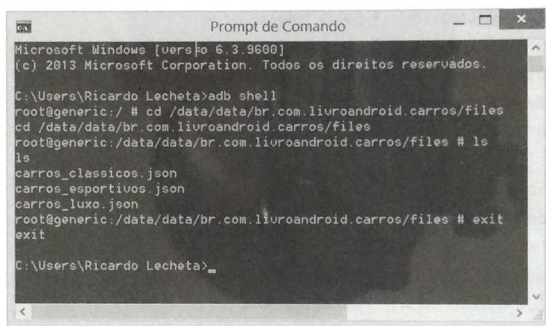
```
adb shell
cd /data/data/br.com.livroandroid.carros/files
ls
```

Dica: lembre-se de que o comando `adb` deve estar no PATH do sistema operacional, ou você deve estar na pasta `/android-sdk/platform-tools/` na qual o `adb` está instalado. No Android Studio, para visualizar o local onde o SDK está instalado, utilize o menu `File > Project Structure > SDK Location`. Aproveite e verifique as outras opções dessa janela, pois há mais informações interessantes nela.

Caso você queira visualizar o conteúdo do arquivo no prompt, digite o comando `cat nome_arquivo`, como por exemplo:

```
cat carros_esportivos.json
```

A figura 18.2 mostra os comandos explicados e os arquivos salvos na memória interna do emulador.



```
Microsoft Windows [vers o 6.3.9600]
(c) 2013 Microsoft Corporation. Todos os direitos reservados.

C:\Users\Ricardo Lecheta>adb shell
root@generic:/ # cd /data/data/br.com.livroandroid.carros/files
cd /data/data/br.com.livroandroid.carros/files
root@generic:/data/data/br.com.livroandroid.carros/files # ls
ls
carros_classicos.json
carros_esportivos.json
carros_luxo.json
root@generic:/data/data/br.com.livroandroid.carros/files # exit
exit

C:\Users\Ricardo Lecheta>_
```

Figura 18.2 – Arquivos salvos no emulador.

Outra maneira de visualizar os arquivos   utilizando a ferramenta **Android Device Monitor**, que pode ser aberta pelo menu **Tools > Android > Android Device Monitor** do Android Studio. Na janela **File Explorer**, ao navegarmos na estrutura de arquivos do emulador, podemos ver que os arquivos estar o l , conforme mostra a figura 18.3. Caso queira copiar o arquivo do emulador para seu computador, utilize o bot o **Pull a File from the Device**, localizado no canto direito superior da janela.

Nota: o emulador permite visualizar todos os arquivos, mas um dispositivo real ter  o acesso a estas pastas bloqueadas por motivos de seguran a. Apenas dispositivos que foram **rooteados** t m acesso a essa pasta. Por m, dar acesso root no Android n o   recomendado, uma vez que isso pode trazer vulnerabilidades de seguran a para o sistema.

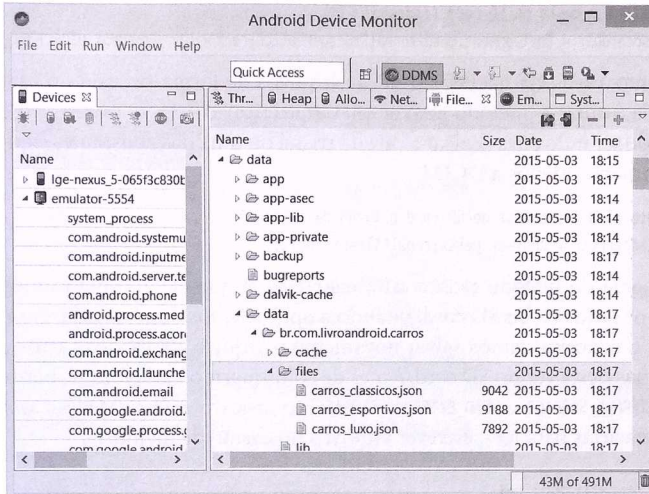


Figura 18.3 – Arquivo salvo na pasta privada da aplicação.

18.5 Trabalhando com arquivos na memória externa (SD card)

O cartão de memória (SD card) é muito utilizado como forma de armazenamento, principalmente para salvar arquivos temporários da aplicação que não precisam ser privados e com isso liberar espaço na memória interna do aparelho. Esse conceito pode ser utilizado, por exemplo, para fazer download de fotos para o SD card.

Para salvar ou ler arquivos no cartão de memória, precisamos obter o objeto `java.io.File` que representa essa pasta, conforme mostra o seguinte código:

```
// Retorna a pasta do SD card
File sdcardDir = android.os.Environment.getExternalStorageDirectory();
```

Depois de obter o objeto `File`, podemos criar pastas e arquivos na raiz do SD card e a partir daqui é pura programação. Mas o recomendado é não poluir o cartão de memória do usuário, principalmente na pasta raiz. O correto é utilizar as pastas padrões para fotos, músicas, downloads etc. Isso pode ser feito com o método `getExternalStoragePublicDirectory(tipo)`, cujo parâmetro `tipo` pode ser qualquer uma destas constantes: `DIRECTORY_MUSIC`, `DIRECTORY_PODCASTS`, `DIRECTORY_RINGTONES`, `DIRECTORY_ALARMS`, `DIRECTORY_NOTIFICATIONS`, `DIRECTORY_PICTURES`, `DIRECTORY_MOVIES`, `DIRECTORY_DOWNLOADS`, ou `DIRECTORY_DCIM`.

```
// Retorna a pasta do SD card (/sdcard/DCIM)
File sdcardDir = Environment.getExternalStoragePublicDirectory(Environment.DIRECTORY_DCIM);
```

Entretanto, caso seja necessário salvar arquivos de forma privada no cartão de memória, utilize o método `getExternalFilesDir(tipo)` da classe `Context`, no qual o tipo pode ser nulo para acessar a raiz da pasta, ou uma das constantes explicadas anteriormente, menos a `DCIM`.

```
// Retorna a pasta raiz do SD card privada da aplicação.
File sdcardDir = context.getExternalFilesDir(null);
```

A vantagem do método `getExternalFilesDir(tipo)` é que ele automaticamente vai excluir os arquivos do SD card, quando a aplicação for desinstalada. Para praticarmos o conceito, vamos salvar novamente o arquivo JSON dos carros em arquivo, mas desta vez no SD card. Antes de continuarmos, adicione as permissões `WRITE_EXTERNAL_STORAGE` e `READ_EXTERNAL_STORAGE` no arquivo *AndroidManifest.xml*, pois são necessárias para ler e escrever arquivos no cartão de memória.

AndroidManifest.xml

```
<manifest ... />
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<application ... />
</manifest>
```

Na sequência, atualize o código-fonte da classe `CarroService`.

CarroService.java

```
...
public class CarroService {
    ...
    public static List<Carro> getCarros(Context context, String tipo) throws IOException {
        String url = URL.replace("{tipo}", tipo);
        String json = HttpHelper.doGet(url);
        salvaArquivoNaMemoriaExterna(context, url, json);
        List<Carro> carros = parserJSON(context, json);
        return carros;
    }
    private static void salvaArquivoNaMemoriaExterna(Context context, String url, String json) {
        String fileName = url.substring(url.lastIndexOf("/") + 1);
        // Cria um arquivo privado
```

```

    File f = SDCardUtils.getPrivateFile(context, fileName, Environment.DIRECTORY_DOWNLOADS);
    IOUtils.writeString(f, json);
    Log.d(TAG, "1) Arquivo privado salvo na pasta downloads: " + f);
    // Cria um arquivo público
    f = SDCardUtils.getPublicFile(fileName, Environment.DIRECTORY_DOWNLOADS);
    IOUtils.writeString(f, json);
    Log.d(TAG, "2) Arquivo público salvo na pasta downloads: " + f);
}
}

```

Feito isso, execute o projeto novamente e você verá as seguintes mensagens no LogCat, indicando que os arquivos foram salvos no SD card:

```

D/CarroService? 1) Arquivo privado salvo na pasta downloads:
/storage/sdcard/Android/data/br.com.livroandroid.carros/files/Download/carros_esportivos.json

D/CarroService? 2) Arquivo público salvo na pasta downloads:
/storage/sdcard/Download/carros_esportivos.json

```

E novamente, se você quiser, é possível visualizar os arquivos com o auxílio da ferramenta adb ou pelo Android Device Monitor. Veja que salvei um arquivo na pasta pública a qual o usuário tem acesso e outro na pasta privada.

Nota: estou utilizando no código as classes `FileUtils` e `SDCardUtils`, dentre outras, para facilitar a explicação. Mas não pense que é porque estou escondendo a complexidade de você. Acredito que essas classes vão facilitar seu aprendizado, até porque todo o código-fonte está disponível no GitHub para sua consulta. No meu dia a dia, é exatamente assim que gosto de programar, pois prefiro encapsular a lógica em classes utilitárias, que resolvem o problema em uma linha de código.

18.6 Outros métodos da classe Context

Outra maneira de obter as principais pastas do sistema para criar arquivos é utilizar os métodos utilitários da classe `Context`, explicados logo a seguir:

Método	Descrição
<code>File getFilesDir()</code>	Retorna o caminho absoluto da pasta em que os arquivos privados são salvos, com o método <code>openFileOutput(String, int)</code> .
<code>File getFileStreamPath(name)</code>	Idem ao anterior, mas retorna o caminho completo do arquivo.

Método	Descrição (cont.)
<code>File getExternalFilesDir(type)</code>	Já explicado anteriormente, e retorna a pasta privada dentro do SD card para o tipo fornecido (DCIM, Downloads etc.).
<code>File getCacheDir()</code>	Retorna uma pasta temporária de sistema ideal para fazer cache de arquivos. Se o sistema ficar com baixa memória, ele pode excluir os arquivos desta pasta para obter mais recursos.

Caso queira brincar com esses métodos, insira o seguinte código dentro de qualquer activity:

```
Log.d("tag", "getFilesDir > " + getFilesDir());
Log.d("tag", "getFilesDir > " + getFilePath("arquivo.txt"));
Log.d("tag", "getFilesDir > " + getExternalFilesDir(Environment.DIRECTORY_DCIM));
Log.d("tag", "getFilesDir > " + getCacheDir());
```

Como resultado, as seguintes mensagens serão impressas no LogCat:

```
D/tag: getFilesDir > /data/data/br.com.livroandroid.carros/files
D/tag: getFilesDir > /data/data/br.com.livroandroid.carros/files/arquivo.txt
D/tag: getFilesDir > /storage/sdcard/Android/data/br.com.livroandroid.carros/files/DCIM
D/tag: getFilesDir > /data/data/br.com.livroandroid.carros/cache
```

18.7 Brincando de fazer cache

O projeto dos carros está buscando os carros do servidor, fazendo uma requisição HTTP no web service que está no site. Mas já que aprendemos a salvar os arquivos, por que não aproveitamos e fazemos um cache?

Caso o JSON dos carros esteja salvo em arquivo, podemos ler o JSON diretamente desse arquivo e evitar uma conexão com a internet, além de melhorar o desempenho do aplicativo. A lógica para a busca será a seguinte:

1. O aplicativo deve fazer a consulta dos carros tentando ler o arquivo JSON que está salvo na memória interna.
2. Se o arquivo existir, faz a leitura e retorna a lista de carros.
3. Se o arquivo não existir, busca os carros no web service e salva o arquivo com o JSON.

Para implementar essa funcionalidade, vamos renomear o método `getCarros()` para `getCarrosFromWebService()`. Pela assinatura, já deu para entender o que este método vai fazer. No seu lugar, vamos criar outro método `getCarros()`.

Feito isso, vamos criar o método `getCarrosFromArquivo()` para criar a listar de carros a partir do arquivo que está salvo na memória interna, caso o arquivo exista, é claro. E o método `getCarros()` vai conter a grande lógica, que consiste em tentar ler os carros do arquivo. Caso o arquivo não exista, vamos buscar os dados do web service. O código-fonte a seguir mostra como fazer essa lógica:



CarroService.java

...

```
public class CarroService {
    public static List<Carro> getCarros(Context context, String tipo) throws IOException {
        List<Carro> carros = getCarrosFromArquivo(context,tipo);
        if(carros != null && carros.size() > 0) {
            // Encontrou o arquivo
            return carros;
        }
        // Se não encontrar, busca no web service
        carros = getCarrosFromWebService(context,tipo);
        return carros;
    }
    // Abre o arquivo salvo, se existir, e cria a lista de carros
    public static List<Carro> getCarrosFromArquivo(Context context, String tipo)
        throws IOException {
        String fileName = String.format("carros_%.json",tipo);
        Log.d(TAG,"Abrindo arquivo: " + fileName);
        // Lê o arquivo da memória interna
        String json = FileUtils.readFile(context,fileName,"UTF-8");
        if(json == null) {
            Log.d(TAG,"Arquivo "+fileName+" não encontrado.");
            return null;
        }
        List<Carro> carros = parserJSON(context, json);
        Log.d(TAG,"Carros lidos do arquivo "+fileName+".");
        return carros;
    }
    // Faz a requisição HTTP, cria a lista de carros e salva o JSON em arquivo
    public static List<Carro> getCarrosFromWebService(Context context, String tipo)
        throws IOException {
        String url = URL.replace("{tipo}", tipo);
        Log.d(TAG,"URL: " + url);
        String json = HttpHelper.doGet(url);
        salvaArquivoNaMemoriaInterna(context, url, json);
    }
}
```

```

        List<Carro> carros = parserJSON(context, json);
        return carros;
    }
    private static void salvaArquivoNaMemoriaInterna(Context context, String url, String json) {
        ...
    }
    private static List<Carro> parserJSON(Context context, String json) throws IOException {}
}

```

Veja que no código estou salvando os arquivos na memória interna, e não alterei nada nessa parte de salvar arquivos, nem no método que faz o parser do JSON. Portanto, se você executar o projeto novamente, tudo deverá continuar funcionando. Caso os arquivos não existam na memória interna, você verá as seguintes mensagens no LogCat:

```

Abrindo arquivo: carros_luxo.json
Arquivo (carros_luxo.json) não encontrado.
URL: http://www.livroandroid.com.br/livro/carros/carros_luxo.json
Arquivo salvo: /data/data/br.com.livroandroid.carros/files/carros_luxo.json

```

E, caso os arquivos já existam, será feito o cache e você verá as seguintes mensagens:

```

Abrindo arquivo: carros_luxo.json
Carros lidos do arquivo carros_luxo.json

```

Nesta brincadeira, você aprendeu a salvar e ler arquivos, tanto na memória interna quanto na memória externa. O resultado até que ficou legal, pois o aplicativo dos carros fez cache dos arquivos, portanto, a lista será carregada mais rapidamente.

Mas o nosso principal objetivo neste capítulo é salvar os carros em um banco de dados, e não em arquivos. Portanto, no próximo tópico vamos começar o nosso estudo sobre SQLite.

18.8 Banco de dados SQLite

O Android tem suporte ao SQLite (<http://www.sqlite.org>), um leve e poderoso banco de dados. Cada aplicação pode criar um ou mais banco de dados que ficam localizados na seguinte pasta:

```
/data/data/pacote.do.aplicativo/databases/
```

Nota: um banco de dados é visível somente à aplicação que o criou.

O banco de dados pode ser criado de várias formas:

- Utilizando a API do Android para o SQLite. Logo depois de criar o banco de dados, é possível executar um script SQL pela API para criar as tabelas e preencher os dados.
- Usando um cliente do SQLite como o **SQLite Expert Personal**, que é gratuito e está disponível para download em: <http://www.sqliteexpert.com/>. Eu também gosto muito do plugin **SQLite Manager** para o Firefox.
- Utilizando o aplicativo **sqlite3** pelo console do emulador. Mais detalhes sobre o **sqlite3** podem ser acessados em <http://www.sqlite.org/sqlite.html>.

Basicamente, podemos dizer que existem duas maneiras de criar o banco de dados no aplicativo. Ou utilizamos a API e escrevemos os scripts SQL para criar as tabelas, ou criamos o banco de dados com uma ferramenta externa e depois importamos o banco de dados já pronto no projeto. Neste último, você precisa escrever algum código que vai copiar o arquivo do banco de dados já pronto para dentro das pastas do aplicativo.

No projeto dos carros, o banco de dados será criado utilizando a primeira opção, que é utilizando a API e executando um script SQL para criar as tabelas necessárias para o aplicativo funcionar.

18.9 Criação de um banco de dados diretamente com a API

No projeto, vamos criar um banco de dados com uma tabela chamada **carro**, conforme este script SQL.

```
create table if not exists carro (_id integer primary key autoincrement,nome text, desc
text, url_foto text,url_info text,url_video text, latitude text,longitude text, tipo text);
```

Eu acredito que a API do Android seja tão simples que a maneira mais fácil de explicar é mostrar logo de uma vez o código-fonte da classe que vamos utilizar. Então crie a classe **CarroDB** no pacote **domain**, conforme demonstrado a seguir. Essa classe contém métodos para listar todos os carros por tipo, salvar ou atualizar um carro, excluir um carro e excluir todos os carros pelo tipo.



CarroDB.java

```
package br.com.livroandroid.carros.domain;
public class CarroDB extends SQLiteOpenHelper {
    private static final String TAG = "sql";
    // Nome do banco
```

```

public static final String NOME_BANCO = "livro_android.sqlite";
private static final int VERSAO_BANCO = 1;
public CarroDB(Context context) {
    // context, nome do banco, factory, versão
    super(context, NOME_BANCO, null, VERSAO_BANCO);
}
@Override
public void onCreate(SQLiteDatabase db) {
    Log.d(TAG, "Criando a Tabela carro...");
    db.execSQL("create table if not exists carro (_id integer primary key
        autoincrement,nome text, desc text, url_foto text,url_info text,url_video text,
        latitude text,longitude text, tipo text);");
    Log.d(TAG, "Tabela carro criada com sucesso.");
}
@Override
public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
    // Caso mude a versão do banco de dados, podemos executar um SQL aqui
}
// Insere um novo carro, ou atualiza se já existe
public long save(Carro carro) {
    long id = carro.id;
    SQLiteDatabase db = getWritableDatabase();
    try {
        ContentValues values = new ContentValues();
        values.put("nome", carro.nome);
        values.put("desc", carro.desc);
        values.put("url_foto", carro.urlFoto);
        values.put("url_info", carro.urlInfo);
        values.put("url_video", carro.urlVideo);
        values.put("latitude", carro.latitude);
        values.put("longitude", carro.longitude);
        values.put("tipo", carro.tipo);
        if (id != 0) {
            String _id = String.valueOf(carro.id);
            String[] whereArgs = new String[]{_id};
            // update carro set values = ... where _id=?
            int count = db.update("carro", values, "_id=?", whereArgs);
            return count;
        } else {
            // insert into carro values (...)
            id = db.insert("carro", "", values);
            return id;
        }
    } finally {

```

```

        db.close();
    }
}
// Deleta o carro
public int delete(Carro carro) {
    SQLiteDatabase db = getWritableDatabase();
    try {
        // delete from carro where _id=?
        int count = db.delete("carro", "_id=?", new String[]{String.valueOf(carro.id)});
        Log.i(TAG, "Deletou [" + count + "] registro.");
        return count;
    } finally {
        db.close();
    }
}
// Deleta os carros do tipo fornecido
public int deleteCarrosByTipo(String tipo) {
    SQLiteDatabase db = getWritableDatabase();
    try {
        // delete from carro where _id=?
        int count = db.delete("carro", "tipo=?", new String[]{tipo});
        Log.i(TAG, "Deletou [" + count + "] registros");
        return count;
    } finally {
        db.close();
    }
}
// Consulta a lista com todos os carros
public List<Carro> findAll() {
    SQLiteDatabase db = getWritableDatabase();
    try {
        // select * from carro
        Cursor c = db.query("carro", null, null, null, null, null, null, null);
        return toList(c);
    } finally {
        db.close();
    }
}
// Consulta o carro pelo tipo
public List<Carro> findAllByTipo(String tipo) {
    SQLiteDatabase db = getWritableDatabase();
    try {
        // "select * from carro where tipo=?"
        Cursor c = db.query("carro", null, "tipo = '" + tipo + "'", null, null, null, null);
        return toList(c);
    }
}

```

```

        } finally {
            db.close();
        }
    }
}

// Lê o cursor e cria a lista de carros
private List<Carro> toList(Cursor c) {
    List<Carro> carros = new ArrayList<Carro>();
    if (c.moveToFirst()) {
        do {
            Carro carro = new Carro();
            carros.add(carro);
            // recupera os atributos de carro
            carro.id = c.getLong(c.getColumnIndex("_id"));
            carro.nome = c.getString(c.getColumnIndex("nome"));
            carro.desc = c.getString(c.getColumnIndex("desc"));
            carro.urlInfo = c.getString(c.getColumnIndex("url_info"));
            carro.urlFoto = c.getString(c.getColumnIndex("url_foto"));
            carro.urlVideo = c.getString(c.getColumnIndex("url_video"));
            carro.latitude = c.getString(c.getColumnIndex("latitude"));
            carro.longitude = c.getString(c.getColumnIndex("longitude"));
            carro.tipo = c.getString(c.getColumnIndex("tipo"));
        } while (c.moveToNext());
    }
    return carros;
}

// Executa um SQL
public void execSQL(String sql) {
    SQLiteDatabase db = getWritableDatabase();
    try {
        db.execSQL(sql);
    } finally {
        db.close();
    }
}

// Executa um SQL
public void execSQL(String sql, Object[] args) {
    SQLiteDatabase db = getWritableDatabase();
    try {
        db.execSQL(sql, args);
    } finally {
        db.close();
    }
}
}

```

Agora vamos explicar um pouco do código. No Android, as classes que precisam criar ou abrir um banco de dados devem herdar de `SQLiteOpenHelper`.

```
public class CarroDB extends SQLiteOpenHelper {
```

Feito isso, no construtor você deve informar o nome do banco de dados e a versão, que é um código numérico que podemos incrementar.

```
    public CarroDB(Context context) {  
        super(context, NOME_BANCO, null, VERSAO_BANCO);  
    }  
}
```

Para utilizar o banco de dados, basta chamar os métodos `getWritableDatabase()` ou `getReadableDatabase()`, que retornam um objeto do tipo `SQLiteDatabase`, o qual representa a conexão com o banco de dados. Caso o banco de dados não exista, o método `onCreate(SQLiteDatabase)` é chamado para executar um script SQL com o objetivo de criar as tabelas no banco de dados.

```
    public void onCreate(SQLiteDatabase db) {  
        db.execSQL("create table if not exists carro (_id integer primary key  
            autoincrement,name text, desc text, url_foto text,url_info text,url_video text,  
            latitude text,longitude text, tipo text);");  
    }  
}
```

Caso a versão do banco de dados informada no construtor seja diferente da versão atual, por exemplo, se você mudou de 1 para 2, o método `onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)` é chamado. Nesse momento, podemos executar algum script para atualizar a versão do banco de dados, seja para criar novas tabelas, adicionar/alterar colunas etc.

```
    public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {  
        // Caso mude a versão do banco de dados, podemos executar um SQL aqui  
        if(oldVersion == 1 && newVersion == 2) {  
            // Mudou a versão...  
        }  
    }  
}
```

Nota: é recomendável que para cada tabela crie-se uma coluna com o nome "`_id`" que seja do tipo autoincremento. Dessa forma, ao inserir o registro o valor do campo "`_id`" é omitido, pois é gerado automaticamente pelo SQLite.

Depois de criar esse template básico, dentro da classe você pode criar os métodos que quiser. A lógica sempre consiste em obter o objeto `SQLiteDatabase`, fazer

alguma operação no banco de dados e depois fechar a conexão. A seguir, temos um template básico de um método qualquer:

```
public class CarroDB extends SQLiteOpenHelper {
    . . .
    public void meuMetodo () {
        SQLiteDatabase db = getWritableDatabase();
        try {
            // Faça o que quiser aqui...
            db.insert(...);
            db.update(...);
            db.delete(...);
            db.execSQL("sql aqui");
        } finally {
            db.close(); // Fecha a conexão.
        }
    }
}
```

Nota: ao chamar o método `getWritableDatabase()`, o Android vai tentar abrir o banco de dados e caso o banco não exista o método `onCreate(db)` será chamado. Nesse momento, você deve executar algum SQL para criar as tabelas. Caso o parâmetro da versão que foi passado no construtor da classe seja diferente da versão atual, o método `onUpgrade(db,oldVersion,newVersion)` será chamado para o aplicativo atualizar o banco de dados.

Embora o código-fonte da classe `CarroDB` esteja completo, nos próximos tópicos vou explicar separadamente cada parte do código, a fim de detalhar um pouco como funciona a API do SQLite.

18.10 Inserção de registros no banco de dados

Para inserir registros no banco de dados, é necessário criar um objeto do tipo `android.content.ContentValues`, que contém uma estrutura simples de chave e valor, em que cada chave precisa corresponder ao nome de uma coluna da tabela.

Depois de criar o objeto `ContentValues` com os valores para inserir o registro, basta chamar o método `insert(tabela,nullColumnHack, contentValues)`. A seguir, podemos visualizar um exemplo de como inserir um carro em uma tabela que contém as colunas nome e descrição.

```
// Insere um novo carro
ContentValues valores = new ContentValues();
valores.put("nome", "Carro");
valores.put("desc", "Descrição");
db.insert("carro", null, valores);
```

Este código é equivalente ao seguinte SQL:

```
insert into table carro(nome,descricao) values ("Carro","Descrição");
```

Veja a explicação dos parâmetros do método insert(...).

Parâmetro	Descrição
String tabela	Nome da tabela.
String nullColumnHack	Nome de uma coluna opcional usada para não permitir que um registro completamente nulo seja inserido. Não utilizaremos esse campo.
ContentValues valores	Estrutura de chave e valores, com os valores para inserir.

18.11 Atualização de registros no banco de dados

Para atualizar um registro, deve-se utilizar o método update(tabela, contentValues, where, whereArgs) e passar uma string para o argumento where do método, em que o valor do id do registro pode ser utilizado para identificar o registro que deve ser atualizado.

O exemplo a seguir mostra como atualizar o carro de id=1:

```
// Cria os dados para atualizar o carro id=1
long id = 1;
ContentValues valores = new ContentValues();
valores.put("nome", "novo nome");
valores.put("desc", "Descrição");

// Atualiza o carro com id=1 com estes valores
db.update("carro", valores, "_id = " + id, null);

// Outra forma é informar os parâmetros no terceiro parâmetro (array)
db.update("carro",valores, "_id=?",new String[]{String.valueOf(id)});
```

Esse código é equivalente ao seguinte SQL:

```
update carro set nome="novo nome", descrição="Descrição" where _id=1;
```

Veja a explicação dos parâmetros do método update(...).

Parâmetro	Descrição
String tabela	Nome da tabela.
ContentValues values	Estrutura de chave e valores, com os valores para atualização.
String where	String com a cláusula <code>where</code> utilizada para identificar o registro. Nesse caso, pode ser uma string com o texto <code>"id=1"</code> , ou uma string com o texto <code>"id=?"</code> , tornando necessário usar o último argumento para informar o valor dos parâmetros.
String whereArgs[]	Array com os parâmetros necessários, caso a cláusula <code>where</code> defina algum parâmetro com <code>"?"</code> .

O código para inserir e atualizar um registro na tabela é simples. Veja que basicamente você precisa preencher o objeto `ContentValues`, em que o nome de cada chave precisa ser exatamente o nome da coluna da tabela.

A classe `ContentValues` facilita o trabalho para inserir ou atualizar registros, pois você não precisa escrever a instrução SQL de cada comando. Caso você quieria fazer manualmente cada SQL, basta utilizar o método `execSQL(sql)`.

18.12 Exclusão de registros do banco de dados

Para remover um registro, deve-se utilizar o método `delete(tabela,where,whereArgs)`. O exemplo de código a seguir mostra como excluir um carro com o `id=1`:

```
// Deleta o carro 1
long id = 1;
// Deleta o carro com id=1
db.delete("carro", "_id=" + id,null);
// Ou informando o terceiro parâmetro
db.delete("carro", "_id=?",new String[]{String.valueOf(id)});
```

Esse código é equivalente ao seguinte SQL:

```
delete from carro where _id=1;
```

18.13 Busca de registros no banco de dados

Para consultar os registros de uma tabela, é utilizado o método `query(...)` informando as colunas desejadas e a cláusula `where` para criar o SQL. O retorno é um objeto do tipo `android.database.Cursor`, que deve ser utilizado para ler os valores da consulta.

Para exemplificar, digamos que você precise executar este SQL:

```
SELECT _id,nome,descricao from carro where _id = ?
```

Pela API, isso pode ser feito com o seguinte código:

```
String id = "1";
Cursor c = db.query("carro", new String[] { "_id", "nome", "descricao" }, "_id=?", new
String[]{id}, null, null, null);
// Se encontrou...
if (c.getCount() > 0) {
    Carro carro = new Carro();
    //Posiciona no primeiro resultado
    c.moveToFirst();
    // Faz a leitura dos valores
    carro.id = c.getLong(0);
    carro.nome = c.getString(1);
    carro.descricao = c.getString(2);
}
```

Veja a explicação dos parâmetros do método `query(...)`.

Parâmetro	Descrição
boolean distinct	Mesmo funcionamento da palavra <code>distinct</code> em um SQL normal. É utilizado para que o resultado não contenha registros repetidos. Esse parâmetro é opcional e existe uma assinatura desse método sem ele.
String tabela	Nome da tabela.
String colunas[]	Array com os nomes das colunas para seleção. Se for passado como parâmetro um array nulo, todas as colunas serão retornadas.
String selecao[]	Contém a cláusula <code>where</code> utilizada para filtrar os registros. Se for informado um parâmetro nulo, todos os registros serão retornados.
String selecaoArgs[]	Argumentos "?" da cláusula <code>where</code> , caso necessário.
String groupBy	Nome das colunas para agrupar (<code>group by</code>).
String orderBy	Nome das colunas para ordenar (<code>order by</code>).

18.14 Métodos da classe `Cursor`

A classe `android.database.Cursor` utilizada para ler os resultados de um registro contém os seguintes métodos:

Método	Descrição
<code>close()</code>	Fecha o cursor, liberando os recursos e a memória.
<code>int getColumnCount()</code>	Retorna o número de colunas da consulta.
<code>int getColumnIndex(String)</code>	Retorna o índice da coluna para o nome de coluna informado. O índice começa em zero e -1 é retornado se a coluna não existir. O índice da coluna é posteriormente utilizado para ler o valor da mesma, usando os métodos <code>getString(i)</code> , <code>getInt(i)</code> , <code>getLong(i)</code> etc.
<code>int getColumnIndexOrThrow(String)</code>	Idem ao método <code>getColumnIndex(String)</code> , mas se não encontrar a coluna, uma exceção <code>IllegalArgumentException</code> é lançada.
<code>String[] getColumnNames</code>	Retorna um array de string com os nomes de colunas disponíveis.
<code>String getColumnName(indice)</code>	Retorna o nome da coluna para o índice especificado.
<code>int getCount()</code>	Retorna o número de registros retornados pela consulta. Funciona de forma similar ao <code>count(*)</code> em SQL.
<code>String getString(i)</code>	Método usado para retornar o valor da coluna no formato string, informando o índice. Podemos ler outros tipos de dados com os métodos <code>getInt(i)</code> , <code>getFloat(i)</code> , <code>getDouble(i)</code> , <code>getShort(i)</code> etc.
<code>boolean moveToFirst()</code>	Posiciona o cursor para leitura no primeiro registro. Método frequentemente utilizado para descobrir se foi retornado algum registro pela consulta, sendo que retorna um valor booleano.
<code>boolean moveToLast()</code>	Posiciona o cursor para leitura no último registro.
<code>boolean moveToNext()</code>	Posiciona o cursor para leitura no próximo registro. Método frequentemente utilizado para ler todos os registros retornados em um loop. Retorna um booleano informando se existe mais um registro para ser lido.
<code>boolean moveToPosition(pos)</code>	Posiciona o cursor para leitura na posição desejada.
<code>boolean moveToPrevious()</code>	Posiciona o cursor para leitura na posição anterior.

Nem todos os métodos estão listados aqui, apenas os principais. Consulte o Javadoc da classe `Cursor` para obter mais informações. Lembre-se de chamar o método `close()` da classe `Cursor` para liberar os recursos do cursor quando ele não for mais necessário.

18.15 Continuando o projeto dos carros

Depois de estudar a API do banco de dados SQLite, vamos continuar o desenvolvimento do projeto dos carros. Nosso objetivo será salvar todos os carros no banco de dados logo depois de consultar a lista no web service.

Na verdade, já fizemos essa lógica de cache, pois os arquivos JSON estão sendo salvos na memória interna. O que vamos fazer agora é simplesmente alterar o código para salvar e ler os carros no banco de dados, substituindo pela versão que salva os arquivos.

Nota: depois de salvar os carros no banco de dados, vamos criar telas para editar e excluir os carros localmente. Assim podemos brincar com os dados do banco, e quando quisermos basta atualizar a lista e puxar os dados do web service.

O código-fonte a seguir mostra como salvar e consultar os carros do banco de dados.



CarroService.java

```
public class CarroService {
    . . .
    public static List<Carro> getCarros(Context context, String tipo) throws IOException {
        List<Carro> carros = getCarrosFromBanco(context, tipo);
        if (carros != null && carros.size() > 0) {
            // Retorna os carros encontrados do banco
            return carros;
        }
        // Se não encontrar, busca no web service
        carros = getCarrosFromWebService(context, tipo);
        return carros;
    }
    public static List<Carro> getCarrosFromBanco(Context context, String tipo)
        throws IOException {
        CarroDB db = new CarroDB(context);
        try {
            List<Carro> carros = db.findAllByTipo(tipo);
            Log.d(TAG, "Retornando " + carros.size() + " carros [" + tipo + "] do banco");
            return carros;
        } finally {
            db.close();
        }
    }
}
```

```

public static List<Carro> getCarrosFromWebService(Context context, String tipo)
    throws IOException {
    String url = URL.replace("{tipo}", tipo);
    Log.d(TAG, "URL: " + url);
    String json = HttpHelper.doGet(url);
    List<Carro> carros = parserJSON(context, json);
    // Depois de buscar salva os carros
    salvarCarros(context, tipo, carros);
    return carros;
}

// Salva os carros no banco de dados
private static void salvarCarros(Context context, String tipo, List<Carro> carros) {
    CarroDB db = new CarroDB(context);
    try {
        // Deleta os carros antigos pelo tipo para limpar o banco
        db.deleteCarrosByTipo(tipo);
        // Salva todos os carros
        for (Carro c : carros) {
            c.tipo = tipo;
            Log.d(TAG, "Salvando o carro " + c.nome);
            db.save(c);
        }
    } finally {
        db.close();
    }
}
}
}

```

Depois dessa alteração, execute o projeto e tudo deverá continuar funcionando. Na primeira execução, o banco de dados será criado e os carros serão salvos. Recomendo você depurar o código para entender o funcionamento e também inserir alguns logs caso você ache necessário. Um bom começo é você olhar os logs do **LogCat**, que mostram as seguintes mensagens na primeira vez que o banco de dados é criado:

```

D/sql? Criando a Tabela carro... // Aqui a classe CarroDB executou o SQL
D/sql? Tabela carro criada com sucesso.
D/CarroService? Retornando 0 (zero) carros [classicos] do banco
D/CarroService? URL: http://www.livroandroid.com.br/livro/carros/carros_classicos.json
D/CarroService? Salvando o carro Tucker 1948
D/CarroService? Salvando o carro Chevrolet Corvette
. . .
D/CarroService? Carros salvos no banco

```

Veja que mostrei os logs apenas dos carros clássicos, mas como o aplicativo abre as três tabs, você verá os logs de todos os carros. E caso você feche o aplicativo e abra novamente, os carros já estarão salvos no banco de dados, nesse caso, você verá os seguintes logs:

D/CarroService:Retornando 10 carros [classicos] do banco.

No próximo tópico, vamos aprender a visualizar o banco de dados que está no emulador.

18.16 Visualizando o banco de dados com a ferramenta adb

Para visualizar o banco de dados criado, vamos utilizar o emulador, pois no dispositivo real por motivos de segurança não é possível acessar a pasta privada do aplicativo. Depois de executar o projeto no emulador, o banco de dados foi criado na pasta `/data/data/br.com.livroandroid.carros/databases`, conforme mostra a figura 18.4.

Isso significa que podemos copiar esse arquivo para o computador para abrir o banco de dados em qualquer cliente de SQLite. Mas antes vamos demonstrar uma técnica bem interessante, a de conectar-se ao emulador com o comando `adb shell`. Lembrando que o comando `adb` deve estar no PATH do sistema operacional, ou você deve estar na pasta `/android-sdk/platform-tools/`, que é onde o `adb` está instalado.

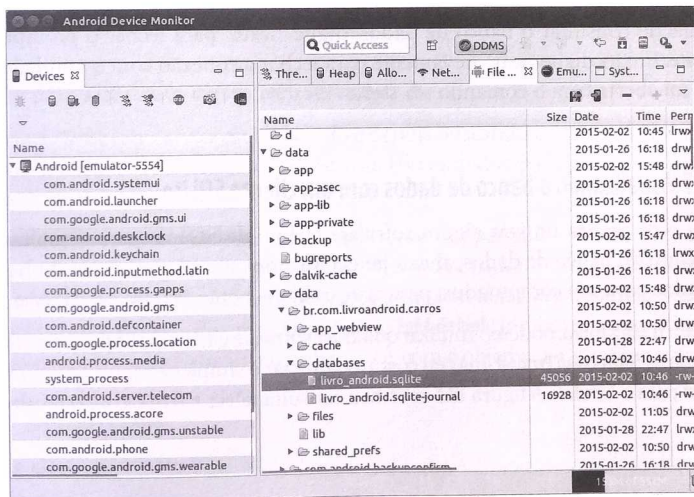


Figura 18.4 – Arquivo do banco de dados.

Dica: no Android Studio, para visualizar o local onde o SDK está instalado utilize o menu File > Project Structure > SDK Location.

Abra um terminal e digite este comando para se conectar ao emulador:

```
adb shell
```

Na sequência, digite o comando `sqlite3 local_do_arquivo_do_banco_de_dados`.

```
sqlite3 /data/data/br.com.livroandroid.carros/databases/livro_android.sqlite
```

Se a conexão for bem-sucedida, o terminal do SQLite estará aberto aguardando um comando SQL.

```
SQLite version 3.8.6 2014-08-15 11:46:33
Enter ".help" for usage hints.
sqlite>
```

Para testar, faça um `SELECT` na tabela dos carros. Veja que deixei apenas três linhas para economizar espaço no livro, mas o `SELECT` retorna todos os carros da tabela.

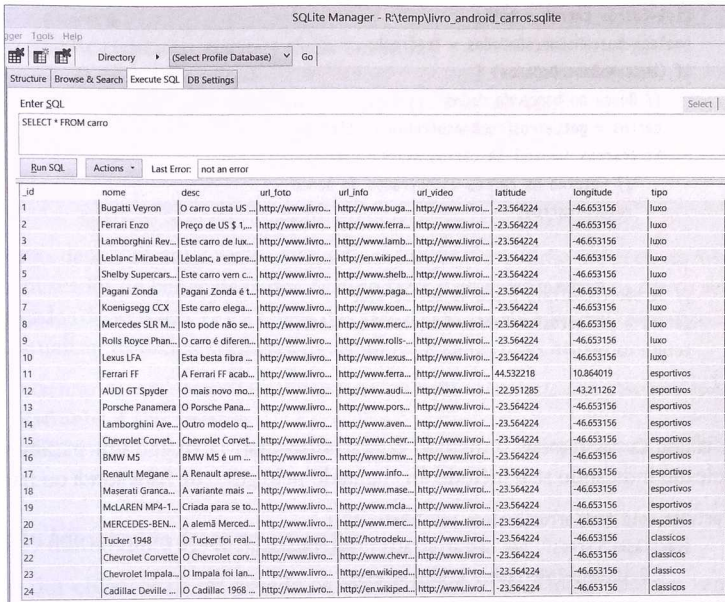
```
sqlite> select _id,nome,tipo from carro;
1|Bugatti Veyron|luxo
2|Ferrari Enzo|luxo
3|Lamborghini Reventon|luxo
```

Depois de consultar o banco de dados, digite `".exit"` para fechar o prompt do SQLite. Então, digite `"exit"` novamente para fechar a conexão com o emulador, a qual foi aberta com o comando `adb shell`.

18.17 Visualizando o banco de dados com um cliente SQLite

Caso você prefira utilizar algum software cliente de SQLite para visualizar o conteúdo do banco de dados, abra a janela **File Explorer** e copie o arquivo do banco de dados para seu computador; para isso, utilize o botão **Pull a File from the device**.

Para abrir o arquivo, podemos utilizar qualquer software cliente do SQLite, como por exemplo o **SQLite Expert Personal**, mas eu costumo utilizar o simples e prático plugin **SQLite Manager** para o Firefox. A figura 18.5 mostra a consulta `select * from carro` e o resultado.



SQLite Manager - R:\temp\livro_android_carros.sqlite

Enter SQL
SELECT * FROM carro

Run SQL Actions Last Error: not an error

_id	nome	desc	url_foto	url_info	url_video	latitude	longitude	tipo
1	Bugatti Veyron	O carro custa US ...	http://www.livro...	http://www.buga...	http://www.livro...	-23.564224	-46.653156	luxo
2	Ferrari Enzo	Prego de US \$ 1...	http://www.livro...	http://www.ferra...	http://www.livro...	-23.564224	-46.653156	luxo
3	Lamborghini Rev...	Este carro de lux...	http://www.livro...	http://www.lamb...	http://www.livro...	-23.564224	-46.653156	luxo
4	Leblanc Mirabeau	Leblanc, a empre...	http://www.livro...	http://en.wikiped...	http://www.livro...	-23.564224	-46.653156	luxo
5	Shelby Supercars	Este carro vern c...	http://www.livro...	http://www.shelb...	http://www.livro...	-23.564224	-46.653156	luxo
6	Pagani Zonda	Pagani Zonda é t...	http://www.livro...	http://www.paga...	http://www.livro...	-23.564224	-46.653156	luxo
7	Koenigsegg CCX	Este carro elega...	http://www.livro...	http://www.koen...	http://www.livro...	-23.564224	-46.653156	luxo
8	Mercedes SLR M...	Isto pode não se...	http://www.livro...	http://www.merc...	http://www.livro...	-23.564224	-46.653156	luxo
9	Rolls Royce Phan...	O carro é diferen...	http://www.livro...	http://www.rolls...	http://www.livro...	-23.564224	-46.653156	luxo
10	Lexus LFA	Esta besta fibra ...	http://www.livro...	http://www.lexus...	http://www.livro...	-23.564224	-46.653156	luxo
11	Ferrari FF	A Ferrari FF acab...	http://www.livro...	http://www.ferra...	http://www.livro...	44.532218	10.864019	esportivos
12	AUDI GT Spyder	O mais novo mo...	http://www.livro...	http://www.audi...	http://www.livro...	-22.951285	-43.211262	esportivos
13	Porsche Panamera	O Porsche Pana...	http://www.livro...	http://www.pors...	http://www.livro...	-23.564224	-46.653156	esportivos
14	Lamborghini Ave...	Outro modelo q...	http://www.livro...	http://www.aven...	http://www.livro...	-23.564224	-46.653156	esportivos
15	Chevrolet Corvet...	Chevrolet Corvet...	http://www.livro...	http://www.chevr...	http://www.livro...	-23.564224	-46.653156	esportivos
16	BMW M5	BMW M5 é um ...	http://www.livro...	http://www.bmw...	http://www.livro...	-23.564224	-46.653156	esportivos
17	Renault Megane ...	A Renault aprese...	http://www.livro...	http://www.info...	http://www.livro...	-23.564224	-46.653156	esportivos
18	Maserati Granca...	A variante mais ...	http://www.livro...	http://www.mase...	http://www.livro...	-23.564224	-46.653156	esportivos
19	McLAREN MP4-1...	Criada para se to...	http://www.livro...	http://www.mcla...	http://www.livro...	-23.564224	-46.653156	esportivos
20	MERCEDES-BEN...	A alemã Merced...	http://www.livro...	http://www.merc...	http://www.livro...	-23.564224	-46.653156	esportivos
21	Tucker 1948	O Tucker foi real...	http://www.livro...	http://hotrodeku...	http://www.livro...	-23.564224	-46.653156	classicos
22	Chevrolet Corvett...	O Chevrolet corv...	http://www.livro...	http://www.chevr...	http://www.livro...	-23.564224	-46.653156	classicos
23	Chevrolet Impala...	O Impala foi lan...	http://www.livro...	http://en.wikiped...	http://www.livro...	-23.564224	-46.653156	classicos
24	Cadillac Deville ...	O Cadillac 1968 ...	http://www.livro...	http://en.wikiped...	http://www.livro...	-23.564224	-46.653156	classicos

Figura 18.5 – Visualizando a tabela carro.

18.18 Banco de dados versus web service

Como a classe `CarroService` está salvando e depois buscando os carros do banco de dados, precisamos ter uma maneira de atualizar os dados, para buscar novamente do web service.

Para solucionar esse problema, vamos adicionar um parâmetro boolean `refresh` no método `CarroService.getCarros(context, tipo, refresh)`. Por padrão, sempre vamos passar `refresh=false` para retornar os carros do banco de dados (caso estejam salvos). Mas se o usuário atualizar a lista de carros com o gesto **Pull to Refresh**, vamos passar o parâmetro `refresh=true` para forçar a busca no web service, com o objetivo de atualizar os dados.



CarroService.java

```
public class CarroService {
    ...
    public static List<Carro> getCarros(Context context, String tipo, boolean refresh)
        throws IOException {
```

```

List<Carro> carros = null;
boolean buscaNoBancoDeDados = !refresh;
if (buscaNoBancoDeDados) {
    // Busca no banco de dados
    carros = getCarrosFromBanco(context, tipo);
    if (carros != null && carros.size() > 0) {
        // Retorna os carros encontrados do banco
        return carros;
    }
}
// Se não encontrar, busca no web service
carros = getCarrosFromWebService(context, tipo);
return carros;
}
}

```

Na classe CarrosFragment, o método taskCarros(boolean) já recebe um parâmetro booleano indicando se o método foi chamado pelo gesto de **Pull to Refresh** ou não.

```

private void taskCarros(boolean pullToRefresh) {
    startTask("carros", new GetCarrosTask(), pullToRefresh
        ? R.id.swipeToRefresh : R.id.progress);
}

```

Portanto, a única coisa que precisamos fazer é repassar esse parâmetro booleano até a classe CarroService, para que lá internamente seja possível decidir se a busca deve ser feita no banco de dados ou no web service.



CarrosFragment.java

```

...
public class CarrosFragment extends BaseFragment {
    ...
    private void taskCarros(boolean pullToRefresh) {
        ...
        startTask("carros", new GetCarrosTask(pullToRefresh),
            pullToRefresh ? R.id.swipeToRefresh : R.id.progress);
    }
    ...
    private class GetCarrosTask implements TaskListener<List<Carro>> {
        private boolean refresh;
        public GetCarrosTask(boolean refresh) {
            this.refresh = refresh;
        }
    }
}

```

```

@Override
public List<Carro> execute() throws Exception {
    // Passa o flag refresh
    return CarroService.getCarros(getContext(), tipo, refresh);
}
...
}
}

```

Pronto, depois de atualizar o código dessa forma, por padrão todos os carros serão buscados do banco de dados. Mas ao fazer o gesto **Pull to Refresh** os carros serão buscados do web service e a tabela dos carros será atualizada com esses registros. Ao atualizar os dados, você verá os seguintes logs no LogCat:

```

D/CarroService:URL: http://www.livroandroid.com.br/livro/carros/carros_classicos.json
D/sql:Deletou [10] registros
D/CarroService: Salvando o carro Tucker 1948 // Aqui vai salvar os outros carros também.

```

18.19 Adicionando ações na action bar

Um dos objetivos de ter salvado os carros no banco de dados, é porque vamos criar telas para editar e excluir os registros. Assim, podemos brincar localmente com os dados, pois podemos restaurar a lista original a qualquer momento.

Crie o seguinte arquivo de menu, que já contém todas as ações que vamos utilizar durante o desenvolvimento do projeto.



/res/menu/menu_frag_carro.xml

```

<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item android:id="@+id/action_video"
        android:icon="@drawable/ic_action_video" android:title="Video"
        app:showAsAction="always" />
    <item android:id="@+id/action_maps"
        android:icon="@drawable/ic_action_map" android:title="Maps"
        app:showAsAction="always" />
    <item android:id="@+id/action_share"
        android:icon="@drawable/ic_action_share" android:title="Share"
        app:showAsAction="ifRoom" />
    <item android:id="@+id/action_edit"
        android:icon="@android:drawable/ic_menu_edit" android:title="Editar"
        app:showAsAction="ifRoom" />

```

```

<item android:id="@+id/action_remove"
      android:icon="@android:drawable/ic_menu_delete" android:title="Remover"
      app:showAsAction="ifRoom" />
</menu>

```

Você vai precisar de algumas imagens, que estão disponíveis nos projetos de exemplo no GitHub do livro. No código da classe, basta inflar o menu do fragment e tratar os eventos. Por enquanto, vamos mostrar um simples toast ao selecionar as opções. Não se esqueça de chamar o método `setHasOptionsMenu(true)`.



CarroFragment.java

```

...
public class CarroFragment extends BaseFragment {
    private Carro carro;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_carro, container, false);
        setHasOptionsMenu(true); // Precisamos informar o Android que este fragment contém menu
        return view;
    }

    ...

    @Override
    public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
        super.onCreateOptionsMenu(menu, inflater);
        inflater.inflate(R.menu.menu_frag_carro, menu);
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        if (item.getItemId() == R.id.action_edit) {
            toast("Editar: " + carro.nome);
            return true;
        } else if (item.getItemId() == R.id.action_remove) {
            toast("Deletar: " + carro.nome);
            return true;
        } else if (item.getItemId() == R.id.action_share) {
            toast("Compartilhar");
        } else if (item.getItemId() == R.id.action_maps) {
            toast("Mapa");
        } else if (item.getItemId() == R.id.action_video) {
            toast("Video");
        }
    }
}

```

```

        return super.onOptionsItemSelected(item);
    }
}

```

A figura 18.6 mostra o resultado com as ações para visualizar o **Vídeo** e **Mapa**, assim como as opções **Share**, **Editar** e **Remover** no menu action overflow.

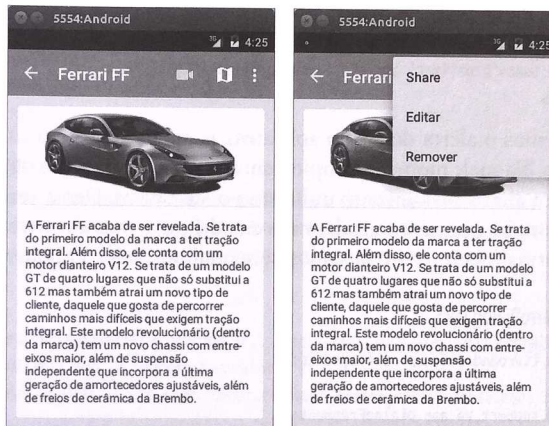


Figura 18.6 – Menus da action bar.

Nota: quando eu me refiro a adicionar ações na action bar, lembre-se de que na prática estamos utilizando a *Toolbar*. O termo action bar virou um padrão de navegação tão famoso que continuamos usando ele, mesmo que na prática a *Toolbar* seja utilizada. A *Toolbar* na verdade é só uma view que permite criar o efeito da action bar, porém com mais flexibilidade.

18.20 Editando o nome do carro

Quando o usuário selecionar a opção **Editar** no menu, vamos mostrar um alerta com um *FragmentManager* para editar o nome do carro. A seguir, podemos ver o layout desse *FragmentManager*.

 /res/layout/fragment_editar_carro.xml

```

<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"

```

```

        android:padding="16dp" android:orientation="vertical">
        <TextView android:text="Nome"
            android:layout_width="match_parent" android:layout_height="wrap_content" />
        <EditText android:id="@+id/tNome"
            android:layout_width="match_parent" android:layout_height="wrap_content"
            android:layout_margin="8dp"/>
        <Button android:id="@+id/btAtualizar"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="Atualizar" android:layout_gravity="right" />
    </LinearLayout>

```

Quando fizemos o alerta de **Sobre** o aplicativo, já aprendemos a utilizar a classe `AlertDialog`. Naquele momento, implementamos o método `onCreateDialog(Bundle)` para criar um alerta customizado utilizando o `AlertDialog`. Dessa vez, como precisamos de um layout customizado, vamos codificar o método `onCreateView()` do fragment normalmente, como faríamos com qualquer outro fragment.



EditarCarroDialog.java

```

package br.com.livroandroid.carros.fragments;
...
import android.support.v4.app.DialogFragment;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentManager;
import android.support.v4.app.FragmentTransaction;
public class EditarCarroDialog extends DialogFragment {
    private Callback callback;
    private Carro carro;
    private TextView tNome;
    // Interface para retornar o resultado
    public static interface Callback {
        public void onCarroUpdated(Carro carro);
    }
    // Método utilitário para criar o dialog
    public static void show(FragmentManager fm, Carro carro, Callback callback) {
        FragmentTransaction ft = fm.beginTransaction();
        Fragment prev = fm.findFragmentByTag("editar_carro");
        if (prev != null) {
            ft.remove(prev);
        }
        ft.addToBackStack(null);
        EditarCarroDialog frag = new EditarCarroDialog();
        frag.callback = callback;
    }
}

```

```

        Bundle args = new Bundle();
        // Passa o objeto carro por parâmetro
        args.putSerializable("carro",carro);
        frag.setArguments(args);
        frag.show(ft, "editar_carro");
    }

    @Override
    public void onStart() {
        super.onStart();
        if (getDialog() == null) {
            return;
        }
        // Atualiza o tamanho do dialog
        int width = getResources().getDimensionPixelSize(R.dimen.popup_width);
        int height = getResources().getDimensionPixelSize(R.dimen.popup_height);
        getDialog().getWindow().setLayout(width, height);
    }

    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
        @Nullable Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_editar_carro, container, false);
        view.findViewById(R.id.btAtualizar).setOnClickListener(onClickAtualizar());
        tNome = (TextView) view.findViewById(R.id.tNome);
        this.carro = (Carro) getArguments().getSerializable("carro");
        if (carro != null) {
            tNome.setText(carro.nome);
        }
        return view;
    }

    private View.OnClickListener onClickAtualizar() {
        return new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                String novoNome = tNome.getText().toString();
                if (novoNome == null || novoNome.trim().length() == 0) {
                    tNome.setError("Informe o nome");
                    return;
                }
                Context context = view.getContext();
                // Atualiza o banco de dados
                carro.nome = novoNome;
                CarroDB db = new CarroDB(context);
                db.save(carro);
            }
        };
    }

```



```

        if(callback != null) {
            callback.onCarroUpdated(carro);
        }
        // Fecha o DialogFragment
        dismiss();
    }
};
}
}
}

```

Lembre-se de importar a classe `android.support.v4.app.DialogFragment` da biblioteca de compatibilidade.

Note que no método `onStart()` do fragment estou configurando o tamanho do alerta com o seguinte código:

```

int width = getResources().getDimensionPixelSize(R.dimen.popup_width);
int height = getResources().getDimensionPixelSize(R.dimen.popup_height);
getDialog().getWindow().setLayout(width, height);

```

Portanto, adicione estas dimensões no projeto:

 `/res/values/dimens.xml`

```

<resources>
    . . .
    <dimen name="popup_width">300dp</dimen>
    <dimen name="popup_height">300dp</dimen>
</resources>

```

Para utilizar o `DialogFragment` que criamos é simples, basta chamar o método utilitário `show()` que encapsula o código para criar o fragment. O mais importante que você precisa entender é que estamos passando como parâmetro um objeto do tipo `EditarCarroDialog.Callback`, que é a interface de retorno (callback) que criei dentro desse fragment. Essa é uma prática muito comum na Orientação a Objetos quando um objeto precisa retornar dados para outro. Essas interfaces fazem o meio de campo entre os objetos. Para continuar, altere o código da classe `CarroFragment` conforme demonstrado a seguir:

 `CarroFragment.java`

```

. . .
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.action_edit) {
        //toast("Editar: " + carro.nome);
    }
}

```

```

    EditarCarroDialog.show(getFragmentManager(), carro, new EditarCarroDialog.Callback() {
        @Override
        public void onCarroUpdated(Carro carro) {
            toast("Carro [" + carro.nome + "] atualizado");
            // Atualiza o título com o novo nome
            getActionBar().setTitle(carro.nome);
        }
    });
    return true;
}
}

```

Observe que internamente no código da classe `EditarCarroDialog` os dados do carro estão sendo salvos, e depois disso o resultado é entregue pelo método `onCarroUpdated(carro)` da interface de callback.

```

// EditarCarroDialog.java
CarroDB db = new CarroDB(context);
db.save(carro);
if(callback != null) {
    callback.onCarroUpdated(carro);
}

```

A figura 18.7 mostra o alerta com o formulário para editar o nome do carro e depois o toast que é mostrado quando o usuário clica no botão **Atualizar**. No código inclusive estamos atualizando o título da action bar para refletir o novo nome do carro.

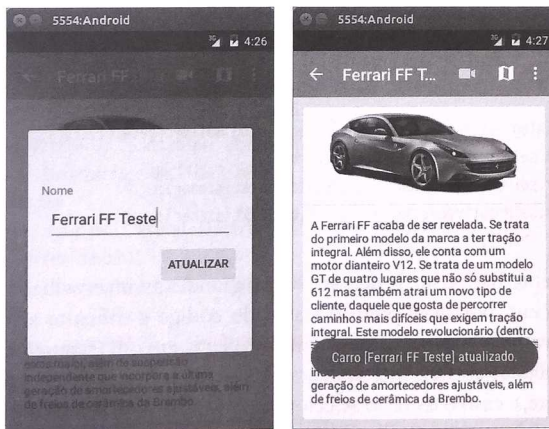


Figura 18.7 – Editando um carro.

Problema: depois de atualizar o nome do carro e voltar para a lista, você verá que o nome antigo ainda está lá. Isso porque precisamos ler novamente os dados para refletir a alteração que fizemos no banco de dados. Então, para ter certeza de que os dados foram atualizados no banco de dados, basta você fechar o aplicativo e abrir novamente. Vamos solucionar esse problema logo, mas antes vamos implementar a ação de deletar.

18.21 Excluindo um carro do banco de dados

Para excluir um carro do banco de dados, vamos mostrar uma alerta para o usuário confirmar a exclusão com as opções **Sim** e **Não**.

Um alerta de confirmação pode ser criado com o seguinte código:

```
DialogInterface.OnClickListener dialogClickListener = new DialogInterface.
OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int which) {
        switch (which){
            case DialogInterface.BUTTON_POSITIVE:
                // Clicou em Sim
                break;
            case DialogInterface.BUTTON_NEGATIVE:
                // Clicou em Não
                break;
        }
    }
};

AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
builder.setMessage("Deletar esse carro?");
builder.setPositiveButton("Sim", dialogClickListener);
builder.setNegativeButton("Não", dialogClickListener);
builder.show();
```

Particularmente, não gosto de deixar códigos grandes assim espalhados na activity ou no fragment, pois isso polui o visual do código e dificulta a manutenção. Portanto, novamente vamos encapsular esse alerta em um `FragmentDialog`. No método `onCreateDialog(Bundle)`, vou utilizar exatamente o trecho de código mostrado anteriormente, e caso o usuário selecione a opção **Sim** vamos retornar o resultado pela interface de retorno (callback).



DeletarCarroDialog.java

```

package br.com.livroandroid.carros.fragments;

public class DeletarCarroDialog extends DialogFragment {
    private Callback callback;
    private Carro carro;
    public static interface Callback {
        public void onCarroDeleted(Carro carro);
    }

    public static void show(FragmentManager fm, Carro carro, Callback callback) {
        FragmentTransaction ft = fm.beginTransaction();
        Fragment prev = fm.findFragmentByTag("deletar_carro");
        if (prev != null) {
            ft.remove(prev);
        }
        ft.addToBackStack(null);
        DeletarCarroDialog frag = new DeletarCarroDialog();
        frag.callback = callback;
        Bundle args = new Bundle();
        args.putSerializable("carro", carro);
        frag.setArguments(args);
        frag.show(ft, "deletar_carro");
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.carro = (Carro) getArguments().getSerializable("carro");
    }

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
        DialogInterface.OnClickListener dialogClickListener = new
            DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                switch (which){
                    case DialogInterface.BUTTON_POSITIVE:
                        // Deleta o carro
                        CarroDB db = new CarroDB(getActivity());
                        db.delete(carro);
                        if(callback != null) {
                            callback.onCarroDeleted(carro);
                        }
                        break;
                }
            }
        };
    }
}

```

```

        case DialogInterface.BUTTON_NEGATIVE:
            break;
    }
}

};
AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
builder.setMessage("Deletar esse carro?");
builder.setPositiveButton("Sim", dialogClickListener);
builder.setNegativeButton("Não", dialogClickListener);
return builder.create();
}
}

```

Na classe `CarroFragment`, ao selecionar a opção para excluir um carro, vamos mostrar o alerta de confirmação. Caso o carro seja excluído, é mostrado um toast, e a activity é encerrada com o método `finish()`.



`CarroFragment.java`

```

...
public boolean onOptionsItemSelected(MenuItem item) {
    ...
    else if (item.getItemId() == R.id.action_remove) {
        DeletarCarroDialog.show(getFragmentManager(), carro, new DeletarCarroDialog.Callback() {
            @Override
            public void onCarroDeleted(Carro carro) {
                toast("Carro [" + carro.nome + "] deletado.");
                // Fecha a activity
                getActivity().finish();
            }
        });
        return true;
    }
}
}

```

A figura 18.8 mostra o alerta com a confirmação para excluir o carro. Ao selecionar a opção **Sim**, o carro é excluído e o aplicativo volta para a tela da lista. Porém, podemos ver que na listagem o carro excluído ainda está lá, pois essa lista precisa ser atualizada para refletir o estado atual do banco de dados.

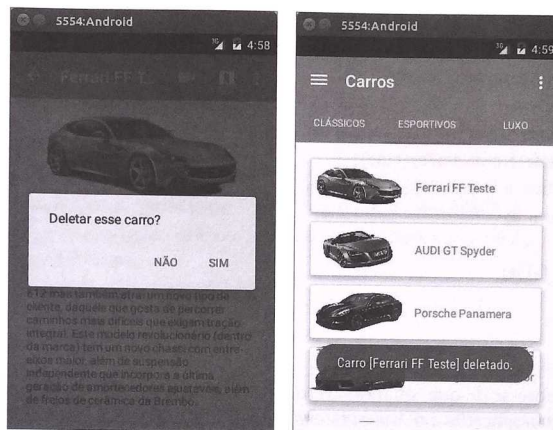


Figura 18.8 – Excluindo o carro.

18.22 Atualizando a lista com dados do web service

Até o momento, temos um problema no aplicativo, pois logo depois de excluir ou editar um carro, ao voltar para a lista, os dados não estão atualizados. Inclusive carros que foram excluídos permanecem na lista.

Na verdade, isso acontece se você clicar no botão de voltar do Android. Porém, se você clicar no up navigation, a lista é atualizada, pois nesse caso é disparada uma intent para a activity pai, fazendo com que ela seja executada novamente. Mas vamos estudar mais detalhes sobre esse comportamento depois.

Para solucionar esse problema, temos de atualizar a lista de carros sempre que fizermos qualquer alteração na tabela do banco de dados. Existem várias formas de fazer isso, e eu vou fazer uma das mais simples, que é utilizar uma variável global no aplicativo, que é um flag booleano que indica se a lista deve ser atualizada. Para isso, altere a classe `CarrosApplication`, que é o singleton que controla a aplicação.



`CarrosApplication.java`

```
...
public class CarrosApplication extends Application {
    // HashMap onde a chave é o tipo do carro e o boolean indica se precisa atualizar
    private Map<String, Boolean> mapUpdate = new HashMap<String, Boolean>();
    ...
}
```

```

public void setPrecisaAtualizar(String tipo, boolean b) {
    // Configura o tipo (clássico,esportivo,luxo) para atualizar a lista
    this.mapUpdate.put(tipo,b);
}
public boolean isPrecisaAtualizar(String tipo) {
    if(mapUpdate.containsKey(tipo)) {
        boolean b = mapUpdate.remove(tipo);
        return b;
    }
    return false;
}
}

```

Nota: lembre-se de que, quando criamos o projeto dos carros, você registrou a classe CarrosApplication no *AndroidManifest.xml*.

Feito isso, nos métodos `onCarroUpdated(carro)` e `onCarroDeleted(carro)`, vamos atualizar este flag para true, assim a lista de carros pode ler o valor desse flag para descobrir se precisa ler os dados novamente. Note que, além do flag booleano, é informado o tipo do carro (clássicos, esportivos ou luxo) para o fragment que controla a lista de carros atualizar a lista do tipo correto.



CarroFragment.java

```

...
public void onCarroUpdated(Carro carro) {
    toast("Carro [" + carro.nome + "] atualizado.");
    CarrosApplication.getInstance().setPrecisaAtualizar(carro.tipo,true);
    // Atualiza o titulo com o novo nome
    getActionBar().setTitle(carro.nome);
}
...
public void onCarroDeleted(Carro carro) {
    toast("Carro [" + carro.nome + "] deletado.");
    CarrosApplication.getInstance().setPrecisaAtualizar(carro.tipo,true);
    // Fecha a activity
    getActivity().finish();
}

```

Feito isso, no método `onResume()` da classe CarrosFragment, vamos testar se houve alterações no banco de dados, e, se for necessário, vamos buscar os carros do banco de dados novamente para atualizar a lista.



CarrosFragment.java

```

...
public class CarrosFragment extends BaseFragment {
    @Override
    public void onResume() {
        super.onResume();
        if(CarrosApplication.getInstance().isPrecisaAtualizar(this.tipo)) {
            // Se teve alterações no banco de dados, vamos atualizar a lista.
            taskCarros(false);
            toast("Lista de carros atualizada!");
        }
    }
}

```

O método `onResume()` foi escolhido para fazer essa verificação, pois ele é executado sempre que a *activity* é exibida para o usuário. Consequentemente, todos os *fragments* também recebem esse evento. Lembre-se de que a tela da lista de carros tem três tabs (clássicos, esportivos e luxo), portanto, esses três *fragments* da classe `CarrosFragment` vão executar o método `onResume()`. Justamente por isso, é passado o tipo do carro para o método `isPrecisaAtualizar(tipo)`, para atualizar a lista correta.

18.23 Modo de execução da *activity* “launchMode”

Ainda existe outro problema no aplicativo que é simples de resolver, então faça comigo o seguinte teste. Role a lista de carros para baixo até o final e selecione o último carro. Feito isso, a tela de detalhes do carro vai aparecer. Ao clicar no botão voltar do Android, a *activity* simplesmente é desempilhada, e a lista de carros volta a aparecer com o seu estado intacto (na posição que estava), ou seja, a lista está com a rolagem lá no final. Mas, se você clicar no botão *up navigation* (seta para esquerda na *action bar*), a rolagem da lista de carros volta para o início, perdendo o estado.

Isso acontece porque o padrão de navegação do *up navigation* é desempilhar todas as *activities* até a raiz. Nesse caso, a *activity* de detalhes `CarroActivity` e a principal `MainActivity` são destruídas e uma nova `MainActivity` é criada. Por isso, perde-se o estado ao clicar no *up navigation*.

Isso acontece porque o modo de execução da *activity*, chamado de *launchMode*, é definido com *standard* por padrão. Para alterar esse comportamento, configure o atributo `android:launchMode="singleTop"` da `MainActivity` no *AndroidManifest.xml*.



AndroidManifest.xml

```
<manifest . . . />
  <application ... />
    <activity
      android:name=".activity.MainActivity" android:launchMode="singleTop" ... />
    ...
</manifest>
```

Feita essa configuração, ao voltar na hierarquia pelo up navigation, a instância da mesma activity será utilizada e não será criada uma nova. Quando isso acontecer, será chamado o método `onNewIntent(intent)` na `MainActivity`, assim a aplicação pode tratar esse evento caso seja necessário. Basicamente, o modo de execução `standard` cria uma nova activity sempre que for disparada uma nova intent com o `startActivity(intent)`. Nesse caso, o método `onCreate(bundle)` é sempre chamado na nova activity. O modo de execução `singleTop` mantém apenas uma instância da activity viva e ao chamar o método `startActivity(intent)` essa activity é colocada no topo da pilha e seu método `onNewIntent(intent)` é chamado.

Este é um assunto avançado, portanto, recomendo que você complemente seus estudos lendo a documentação oficial do Android quando achar necessário. Vale a pena procurar pelo termo `FLAG_ACTIVITY_CLEAR_TOP`, pois ele é muito útil, uma vez que você pode iniciar uma activity limpando o topo da pilha.

<http://developer.android.com/guide/components/tasks-and-back-stack.html>

<http://developer.android.com/guide/topics/manifest/activity-element.html#lmode>

18.24 Fazendo backup na nuvem

Outra forma interessante de persistência disponível no Android é a API de backup. Com ela, podemos salvar pequenas informações de forma transparente na nuvem (cloud) do Google e restaurá-las futuramente. Isso é útil nos casos em que o usuário troca de aparelho, pois, ao configurar o novo celular, é possível restaurar os dados dos aplicativos.

Para fazer o backup dos dados, podemos utilizar a classe `BackupAgent`, na qual precisamos implementar os métodos `onBackup()` e `onRestore()`, responsáveis por transferir os dados do servidor para a nuvem e restaurar o backup posteriormente. Mas implementar essa tarefa não é trivial e exige algumas linhas de código. Por esse motivo, existe a classe `BackupAgentHelper`, que é filha de `BackupAgent` e faz todo esse trabalho. Em conjunto com a classe `BackupAgentHelper`, podemos utilizar outras duas classes para

automatizar o processo de backup. A primeira delas é a `SharedPreferencesBackupHelper`, capaz de fazer backup do que foi salvo nas preferências do usuário, e a segunda é a `FileBackupHelper`, utilizada para fazer backup de arquivos. No aplicativo dos carros, estamos salvando nas preferências do usuário `SharedPreferences` a tab selecionada, portanto vamos aprimorar o aplicativo e salvar isso na nuvem. Para começar a brincadeira, crie a classe `ExemploBackupAgent`. Atenção para o pacote que você vai utilizar, pois será necessário configurar essa classe no *AndroidManifest.xml*.



ExemploBackupAgent.java

```
package br.com.livroandroid.carros.backup;
. . .
public class ExemploBackupAgent extends BackupAgentHelper {
    @Override
    public void onCreate() {
        // Cria um helper. Fará backup dos dados utilizando a chave Prefs.PREF_ID.
        SharedPreferencesBackupHelper helper = new
            SharedPreferencesBackupHelper(this, Prefs.PREF_ID);
        // Adiciona o helper ao agente de backups
        addHelper("livroAndroid", helper);
    }
    @Override
    public void onBackup(ParcelFileDescriptor oldState, BackupDataOutput data,
        ParcelFileDescriptor newState) throws IOException {
        super.onBackup(oldState, data, newState);
        Log.d("backup", "Backup efetuado com sucesso.");
    }
    @Override
    public void onRestore(BackupDataInput data, int appVersionCode, ParcelFileDescriptor newState)
        throws IOException {
        super.onRestore(data, appVersionCode, newState);
        Log.d("backup", "Backup restaurado com sucesso.");
    }
}
```

Essa classe precisa implementar apenas o método `onCreate()` e adicionar quantos helpers forem necessários com o método `addHelper(helper)`. Neste caso, estamos utilizando a classe `SharedPreferencesBackupHelper` a fim de copiar todos os dados salvos na preferência do usuário para a nuvem do Google. Justamente por isso, em seu construtor é informada a chave das preferências, que, neste caso, é a string `Prefs.PREF_ID`, conforme podemos visualizar neste trecho de código:

```
SharedPreferencesBackupHelper helper = new SharedPreferencesBackupHelper(this, Prefs.PREF_ID);
```

A string `Prefs.PREF_ID` é utilizada, pois queremos fazer backup dos dados que foram salvos com a classe `Prefs`. Internamente, se você olhar o código-fonte da classe `Prefs`, ela salva todas as informações utilizando essa chave "livroandroid".

```
public class Prefs {
    public static final String PREF_ID = "livroandroid";
}
```

Uma vez que a classe que fará o backup está criada, precisamos adicionar as tags `android:allowBackup` e `android:backupAgent` ao arquivo *AndroidManifest.xml*.

- A tag `android:allowBackup` recebe um booleano indicando que o aplicativo faz backup, portanto, neste caso, deve ser `true`.
- A tag `android:backupAgent` recebe o nome da classe responsável pelo backup.

Baseado nessas informações, altere o arquivo *AndroidManifest.xml* conforme demonstrado no código a seguir:



AndroidManifest.xml

```
<manifest . . . >
    <application android:allowBackup="true"
        android:backupAgent=".backup.ExemploBackupAgent" . . . >
        <!-- Activities aqui... -->
        <!-- Chave do backup service do Google -->
        <meta-data android:name="com.google.android.backup.api_key"
            android:value="AEdPqrEAAAAI3RuVsUKnTxDy1Xr3YwRUdD0j9p-tEsebzArJZg" />
    </application>
</manifest>
```

Observe que também adicionei uma tag `<meta-data>`, que declara a chave de autenticação do servidor de backup utilizado pelo Google:

```
<!-- Chave do serviço de backup do Google -->
<meta-data android:name="com.google.android.backup.api_key"
    android:value="AEdPqrEAAAAI3RuVsUKnTxDy1Xr3YwRUdD0j9p-tEsebzArJZg" />
```

O valor dessa chave é único para sua aplicação e deve ser gerado nesta página:

<https://developer.android.com/google/backup/signup.html>

Ao abrir a página, você deve ler e aceitar a licença e preencher o campo com o nome do pacote do seu projeto, que, neste caso, é `br.com.livroandroid.carros`. O resultado disso é que a chave necessária para a tag `<meta-data>` será exibida na tela e você poderá copiar o valor para seu projeto.

Pronto, estamos quase lá! A configuração do backup já está feita, e agora falta apenas adicionarmos uma linha de código para executar o backup sempre que alguma informação for salva nas preferências. Com esse objetivo, altere a classe `CarrosTabFragment` desta forma:



CarrosTabFragment.java

```
public class CarrosTabFragment extends BaseFragment {
    private BackupManager backupManager;
    . . .
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        // Gerenciador de backup
        backupManager = new BackupManager(getContext());
        . . .
        mSlidingTabLayout.setOnPageChangeListener(new ViewPager.OnPageChangeListener() {
            @Override
            public void onPageSelected(int position) {
                // Salva o índice da página/tab selecionada
                Prefs.setInteger(getContext(), "tabIdx", viewPager.getCurrentItem());
                backupManager.dataChanged(); // Faz o backup
            }
            . . .
        });
    }
}
```

A classe responsável por disparar o processo de backup é a classe `BackupManager`, por isso, logo depois de alterarmos a tab selecionada, estamos chamando o método `dataChanged()`. Feito isso, o Android vai disparar a classe de agente de backup que está registrada no *AndroidManifest.xml*, portanto, o método `onBackup()` da classe `ExemploBackupAgent` será chamado. E felizmente, como utilizamos a classe `SharedPreferencesBackupHelper` para fazer o backup das preferências, ela já faz tudo automaticamente.

Para testar o backup, execute o projeto mais uma vez e clique em alguma tab.

Como o aplicativo ainda não está publicado no Google Play, precisamos executar um comando para fazer o backup. Portanto, abra um prompt e digite os seguintes comandos. Lembrando que a ferramenta `adb` fica na pasta `/android-studio/sdk/platform-tools`.

```
adb shell bmgr enable true
Backup Manager now enabled
adb shell bmgr backup br.com.livroandroid.carros
adb shell bmgr run
adb uninstall br.com.livroandroid.carros
Success
```

Esses comandos disparam o backup no emulador e logo depois desinstalam o aplicativo, simulando que o backup foi realizado. Nos logs do LogCat, inclusive você poderá ver algumas mensagens como essas:

```
V/BackupServiceBinder? doBackup() invoked
D/backup? Backup efetuado com sucesso.
```

E, para testar se a teoria funciona, execute o projeto novamente. Se tudo funcionar conforme o esperado, você verá que a tab selecionada será aquela tab que você salvou. No momento da restauração do backup, você verá as seguintes mensagens no LogCat:

```
V/BackupServiceBinder? doRestore() invoked
D/backup? Backup restaurado com sucesso.
```

Nota: caso o aplicativo seja instalado pelo Google Play, naturalmente não é necessário executar esses comandos para efetuar o backup. Fizemos isso porque estamos no ambiente de desenvolvimento.

18.25 Fazendo backup de um arquivo

No exemplo anterior, fizemos o backup das preferências do usuário, mas, se quisermos, também podemos fazer backup de arquivos. É importante ressaltar que, devido a possíveis restrições de velocidade de conexão, é recomendado não fazer backup de arquivos grandes.

Como anteriormente no projeto dos carros, salvamos os arquivos JSON na memória interna; se quisermos, é possível fazer o backup desses arquivos.

```
/data/data/br.com.livroandroid.carros/files/carros_classicos.json
```

O código a seguir mostra como fazer backup de um arquivo que está salvo na memória interna. Note que é utilizada a classe `FileBackupHelper`.

**ExemploBackupAgent.java**

```
public class ExemploBackupAgent extends BackupAgentHelper {
    @Override
    public void onCreate() {
        . . .
        // Faz backup de um arquivo
        FileBackupHelper file = new FileBackupHelper(this, "carros_classicos.json");
        addHelper("livroAndroid-file", file);
    }
}
```

A classe `FileBackupHelper` faz backup somente de arquivos salvos na memória interna. Caso você precise salvar um arquivo que está no SD card ou qualquer outra informação, crie sua própria classe agente de backup e implemente os métodos `onBackup()` e `onRestore()`. Se isso for necessário, consulte a documentação oficial do Android.

Lembrando que, se essa solução não lhe servir por algum motivo, existem diversos serviços e APIs que podem ser utilizados para fazer backup de dados, como Google Drive, Amazon S3, Dropbox etc.

18.26 Links úteis

Neste capítulo, estudamos mecanismos de persistência do Android. Para continuar seus estudos, separei alguns links interessantes.

- **Android Training – Saving Key-Value Sets**
<http://developer.android.com/training/basics/data-storage/shared-preferences.html>
- **Android API Guides – Settings**
<http://developer.android.com/guide/topics/ui/settings.html>
- **Android Training – Saving Files**
<http://developer.android.com/training/basics/data-storage/files.html>
- **Android Training – Saving Data in SQL Databases**
<http://developer.android.com/training/basics/data-storage/databases.html>
- **Android Training – Backup API**
<http://developer.android.com/training/cloudsync/backupapi.html>
- **Android Developers – Explicação sobre o atributo `launchMode=singleTop`**
<http://developer.android.com/guide/components/tasks-and-back-stack.html>



CAPÍTULO 19

Action bar de contexto e compartilhamento

Neste capítulo, vamos aprender a utilizar a *contextual action bar*, mas vamos chamá-la apenas de action bar de contexto ou simplesmente CAB.

Também vamos praticar a ação de compartilhamento para enviar informações como texto ou fotos para outras aplicações.

19.1 Introdução

Geralmente, em telas de listagem, o usuário está acostumado a tocar em uma linha e segurar por mais ou menos dois segundos. Feito isso, o sistema geralmente mostra algumas opções com um menu flutuante ou às vezes até pode criar uma action bar de contexto.

A figura 191, extraída da documentação oficial do Android, mostra duas opções comuns ao tocar e segurar na lista por mais de dois segundos. No lado esquerdo, temos o Floating Context Menu e na direita a Contextual Action Bar (CAB).

Criar a primeira opção é relativamente simples, e tenho certeza de que, se você olhar a documentação oficial, vai conseguir fazer. Mas criar a action bar de contexto requer um pouco mais de trabalho, por isso vamos estudá-la. A action bar de contexto (CAB) é muito utilizada em diversos aplicativos nativos do Android. Com certeza, ao utilizá-la, deixaremos nossa aplicação mais refinada, e o usuário vai gostar, uma vez que esse é um comportamento com o qual ele está acostumado e espera que seu aplicativo comporte-se da mesma maneira. Por exemplo, o Gmail permite selecionar várias mensagens ao mesmo tempo para excluí-las, e a Galeria de Fotos permite selecionar várias fotos para compartilhá-las. Vamos implementar exatamente essa funcionalidade no aplicativo dos carros, para permitir que vários carros possam ser excluídos ou compartilhados.

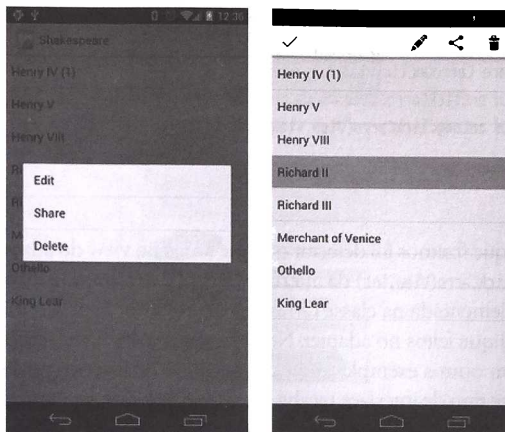


Figura 19.1 – Opções de menu.

19.2 Detectando toques longos – OnLongClickListener

Para começar a brincadeira, precisamos detectar o toque longo na view do adapter, e isso é feito com o método `setOnLongClickListener(listener)`. Então vamos lá, atualize o código da classe `CarroAdapter` conforme demonstrado a seguir:



CarroAdapter.java

```
public class CarroAdapter extends RecyclerView.Adapter<CarroAdapter.CarrosViewHolder> {
    . . .
    @Override
    public void onBindViewHolder(final CarrosViewHolder holder, final int position) {
        . . .
        // Click normal
        if (carroOnClickListener != null) {
            holder.itemView.setOnClickListener(...);
        }
        // Click longo
        holder.itemView.setOnLongClickListener(new View.OnLongClickListener() {
            @Override
            public boolean onLongClick(View v) {
                carroOnClickListener.onLongClickCarro(holder.itemView, position);
                return true;
            }
        })
    }
}
```



```

    });
}
public interface CarroOnClickListener {
    public void onClickCarro(View view, int idx);
    public void onLongClickCarro(View view, int idx);
}
...
}

```

Basicamente o que fizemos foi detectar o toque longo na view do adapter e chamar o método `onLongClickCarro(View,int)` da interface `CarroOnClickListener`. Lembrando que essa interface é implementada na classe `CarrosFragment` e por meio dela o fragment recebe os eventos de clique feitos no adapter. Novamente essa é a interface de callback que já utilizamos em outros exemplos, mas aqui chamei de listener, pois é outro nome comum que esse tipo de interface recebe. Para terminar essa configuração, atualize o código da classe `CarrosFragment` para implementar o método `onLongClickCarro(view,idx)`.

 `CarrosFragment.java`

```

...
private CarroAdapter.CarroOnClickListener onClickCarro() {
    return new CarroAdapter.CarroOnClickListener() {
        @Override
        public void onClickCarro(View view, int idx) { ... }
        @Override
        public void onLongClickCarro(View view, int idx) {
            Carro c = carros.get(idx);
            toast("Clicou e segurou: " + c.nome);
        }
    };
}
}

```

A figura 19.2 mostra o resultado ao tocar em um carro e segurar por dois segundos.

19.3 Ativando o ActionMode na action bar

Quando o usuário tocar em algum carro da lista e segurar por alguns segundos, vamos habilitar a action bar de contexto para mostrar as opções para excluir ou compartilhar vários carros.

Nesse momento, será possível seleccionar vários carros ao mesmo tempo, conforme mostra a figura 19.3. Na figura também podemos ver que foi feito o compartilhamento dos carros seleccionados, e os nomes foram passados por SMS.

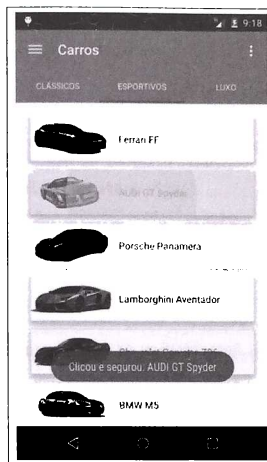


Figura 19.2 – OnLongClickListener na lista.

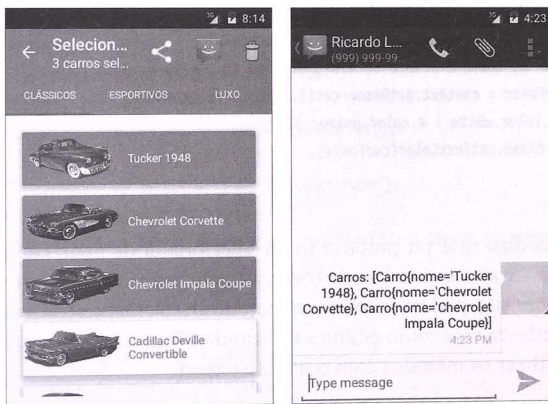


Figura 19.3 – Action bar de contexto (CAB).

Ao selecionar os carros na lista, precisamos de um flag para controlar se o carro está selecionado ou não. Se estiver, vamos pintar a linha da lista de azul ou outra cor. Para criar o flag, altere a classe Carro conforme mostra este código:

**Carro.java**

```

...
public class Carro implements Serializable {
    ...
    public boolean selected; // Flag para indicar que o carro está selecionado
}

```

Para pintar a linha de azul caso o carro esteja selecionado, altere o código do adapter:

**CarroAdapter.java**

```

public class CarroAdapter extends RecyclerView.Adapter<CarroAdapter.CarrosViewHolder> {
    ...
    @Override
    public void onBindViewHolder(final CarrosViewHolder holder, final int position) {
        ...
        // Pinta o fundo de azul se a linha estiver selecionada
        int corFundo = context.getResources().getColor(c.selected
            ? R.color.primary : R.color.white);
        holder.cardView.setCardBackgroundColor(corFundo);
        // A cor do texto é branca ou azul, depende da cor do fundo.
        int corFonte = context.getResources().getColor(c.selected
            ? R.color.white : R.color.primary);
        holder.tNome.setTextColor(corFonte);
    }
}

```

Já temos o código que vai pintar a linha selecionada de azul, então chegou o momento de ativar a action bar de contexto para permitir selecionar os carros. Com esse propósito, podemos utilizar o método `startActionMode(callback)` da classe `android.app.Activity`, mas como estamos utilizando a biblioteca de compatibilidade v7, vamos utilizar os métodos com o prefixo support.

**CarrosFragment.java**

```

...
import android.support.v7.view.ActionMode;
public class CarrosFragment extends BaseFragment {
    private ActionMode actionMode;
    ...
    private CarroAdapter.CarroOnClickListener onClickCarro() {
        return new CarroAdapter.CarroOnClickListener() {

```

```

@Override
public void onClickCarro(View view, int idx) { . . . }
@Override
public void onLongClickCarro(View view, int idx) {
    if (actionMode != null) { return; }
    // Liga a action bar de contexto (CAB)
    actionMode = getCompatActivity().
        startSupportActionMode(getActionModeCallback());
    Carro c = carros.get(idx);
    c.selected = true; // Seleciona o carro
    // Solicita ao Android para desenhar a lista novamente
    recyclerView.getAdapter().notifyDataSetChanged();
    // Atualiza o título para mostrar a quantidade de carros selecionados
    updateActionModeTitle();
}
};
}
}
// Atualiza o título da action bar (CAB)
private void updateActionModeTitle() {
    if (actionMode != null) {
        actionMode.setTitle("Selecione os carros.");
        actionMode.setSubtitle(null);
        List<Carro> selectedCarros = getSelectedCarros();
        if (selectedCarros.size() == 1) {
            actionMode.setSubtitle("1 carro selecionado");
        } else if (selectedCarros.size() > 1) {
            actionMode.setSubtitle(selectedCarros.size() + " carros selecionados");
        }
    }
}
// Retorna a lista de carros selecionados
private List<Carro> getSelectedCarros() {
    List<Carro> list = new ArrayList<Carro>();
    for (Carro c : carros) {
        if (c.selected) { list.add(c); }
    }
    return list;
}
}
}

```

Dica: o método `notifyDataSetChanged()` do adapter tem como objetivo invalidar a lista, forçando o Android a redesenhar a lista. Isso é utilizado para redesenhar a view quando você adiciona ou remove itens da lista, ou deseja alterar a cor do carro selecionado, como é o nosso caso.

Esta linha de código inicia o modo de ação (action mode), que na prática significa que vamos mostrar a action bar de contexto (CAB).

```
actionMode = getCompatActivity().startSupportActionMode(getActionModeCallback());
```

Observe que o objeto `ActionMode` retornado é salvo como um atributo da classe, pois precisamos saber se a action bar de contexto já foi ligada ou não.

Nota: nas próximas citações, para simplificar o texto, vou chamar o termo action bar de contexto apenas de CAB, pois é a sigla para contextual action bar na documentação oficial.

Mas para o código compilar precisamos criar o método `getActionModeCallback()`, que vai retornar um objeto do tipo `ActionMode.Callback`, o qual contém os métodos para inflar as opções de menu que a CAB vai ter, além de tratar os eventos de clique nestas ações.



CarrosFragment.java

```
...
public class CarrosFragment extends BaseFragment {
    ...
    private ActionMode.Callback getActionModeCallback() {
        return new ActionMode.Callback() {
            @Override
            public boolean onCreateActionMode(ActionMode mode, Menu menu) {
                // Infla o menu específico da action bar de contexto (CAB)
                MenuInflater inflater = getActivity().getMenuInflater();
                inflater.inflate(R.menu.menu_frag_carros_context, menu);
                return true;
            }
            @Override
            public boolean onPrepareActionMode(ActionMode mode, Menu menu) {
                return true;
            }
            @Override
            public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
```

```

        List<Carro> selectedCarros = getSelectedCarros();
        if (item.getItemId() == R.id.action_remove) {
            toast("Remover " + selectedCarros);
        } else if (item.getItemId() == R.id.action_share) {
            toast("Compartilhar: " + selectedCarros);
        }
        // Encerra o action mode
        mode.finish();
        return true;
    }

    @Override
    public void onDestroyActionMode(ActionMode mode) {
        // Limpa o estado
        actionMode = null;
        // Configura todos os carros para não selecionados
        for (Carro c : carros) {
            c.selected = false;
        }
        recyclerView.getAdapter().notifyDataSetChanged();
    }
}
};
}
}

```

Para o código compilar, crie o arquivo de menu que a CAB vai inflar.

```

📄 /res/menu/menu_frag_carros_context.xml

<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/action_share" android:icon="@drawable/ic_action_share"
        android:title="Share" app:showAsAction="always" />
    <item
        android:id="@+id/action_remove" android:icon="@android:drawable/ic_menu_delete"
        android:title="Remover" app:showAsAction="always" />
</menu>

```

Também será preciso adicionar mais duas propriedades no arquivo *styles.xml*, responsável por configurar o tema da aplicação. Configure o atributo `windowActionBarOverlay` para `true`, a fim de posicionar a CAB por cima da Toolbar. Você pode testar o aplicativo com e sem essa linha para ver a diferença. Outro atributo necessário é o `actionModeBackground`, que define a cor da CAB, e neste caso deixarei com a cor primária do aplicativo.



/res/values/styles.xml

```

<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
        . . .
        <!-- Action Mode Overlay -->
        <item name="windowActionModeOverlay">true</item>
        <!-- Cor da action bar de contexto (CAB) -->
        <item name="actionModeBackground">@color/primary</item>
    </style>
    . . .
</resources>

```

Não se assuste com o tamanho do código para criar o CAB, pois se você olhar os métodos da interface `ActionMode.Callback` eles são simples. O método `onCreateActionMode()` é responsável por inflar o menu da CAB. Neste menu, vamos mostrar as ações para **Compartilhar** ou **Remover** os carros. O método `onPrepareActionMode()` é chamado sempre que o menu da CAB é invalidado, mas neste exemplo não vamos utilizá-lo. O método `onOptionsItemSelected()` é chamado ao clicar em algumas das opções da CAB e funciona da mesma forma que o famoso método `onOptionsItemSelected(item)` que trata os eventos das ações da action bar. O método `onDestroyActionMode()` é chamado quando a CAB é encerrada, e neste momento devemos limpar os recursos, pois a action bar deve voltar ao seu estado padrão. Para encerrar a CAB, deve-se chamar o método `actionMode.finish()`. Um bom momento para fazer isso é depois de o usuário escolher alguma das opções.

Neste momento, execute o projeto para conferir o resultado. Ao selecionar um carro por mais de dois segundos, a CAB será ativada. No entanto, você deve ter percebido um bug ao ativar a CAB, pois somente o primeiro carro é selecionado. Se você tocar no segundo carro, é feita a navegação para a activity de detalhes, em vez de selecionar a linha.

Isso acontece porque o método `onClickCarro(view, idx)` está programado para sempre fazer a navegação de telas.

```

public void onClickCarro(View view, int idx) {
    Carro c = carros.get(idx);
    Intent intent = new Intent(getContext(), CarroActivity.class);
    intent.putExtra("carro", c);
    startActivity(intent);
}

```

Para resolver esse problema, altere o método `onClickCarro(view, idx)` para testar se a CAB está ativada; neste caso, o correto é apenas selecionar o carro e redesenhar a lista.



CarrosFragment.java

```

public void onClickCarro(View view, int idx) {
    Carro c = carros.get(idx);
    if (actionMode == null) {
        Intent intent = new Intent(getContext(), CarroActivity.class);
        intent.putExtra("carro", c);
        startActivity(intent);
    } else { // Se a CAB está ativada
        // Seleciona o carro
        c.selected = !c.selected;
        // Atualiza o título com a quantidade de carros selecionados
        updateActionModeTitle();
        // Redesenha a lista
        recyclerView.getAdapter().notifyDataSetChanged();
    }
}

```

Com essa alteração, a seleção múltipla de linhas vai funcionar quando a CAB estiver ativada. Caso contrário, a navegação para a tela de detalhes do carro é feita normalmente.

19.4 Removendo os carros selecionados

Uma vez que a CAB estiver ativada, vamos tratar da ação de deletar da action bar.

Como já criamos a classe CarroDB, vamos apenas recuperar a lista de carros selecionados e chamar o método `deleteCarro(c)` para cada objeto.



CarrosFragment.java

```

@Override
public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
    List<Carro> selectedCarros = getSelectedCarros();
    if (item.getItemId() == R.id.action_remove) {
        CarroDB db = new CarroDB(getContext());
        try {
            for (Carro c: selectedCarros) {
                db.delete(c); // Deleta o carro do banco
                carros.remove(c); // Remove da lista
            }
        } finally {
            db.close();
        }
    }
}

```



```

    }
    toast("Carros excluídos com sucesso.");
} else if (item.getItemId() == R.id.action_share) {
    toast("Compartilhar: " + selectedCarros);
}
mode.finish();// Encerra o action mode
return true;
}

```

Concluídas essas alterações, você poderá ativar a CAB para excluir vários carros de uma única vez. Depois de brincar um pouco com isso, recomendo fazer o gesto de **Pull to Refresh** para buscar os carros do web service, e restaurar a lista original.

Um padrão de interface bem comum no Material Design é uma view quadrada com uma mensagem que aparece na parte inferior da tela. Essa view é chamada de *Snackbar*, e um exemplo de onde ela é utilizada é o aplicativo do Gmail. No Gmail, depois de selecionar e excluir um email, uma mensagem aparece com o texto **1 deleted** e a ação **UNDO** para desfazer a exclusão. A boa notícia é que a *Snackbar* faz parte da *Android Design Support Library*, que pode ser configurada no projeto por meio desta dependência:

 app/build.gradle

```

...
compile 'com.android.support:design:22.2.0'

```

O código para mostrar uma *Snackbar* é similar ao utilizado para mostrar um *Toast*, conforme demonstrado a seguir. Observe que o primeiro parâmetro é a view que serve como âncora para mostrar a mensagem; neste caso utilizamos o *RecyclerView*.

```

Snackbar.make(recyclerView, "Carros excluídos com sucesso.", Snackbar.LENGTH_LONG)
    .setAction("OK", new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            toast("OK");
        }
    }).show();

```

19.5 Compartilhando os carros selecionados

Para compartilhar os carros selecionados, altere o arquivo de menu que é inflado para criar a CAB e configure a classe `android.support.v7.widget.ShareActionProvider`.

 /res/menu/menu_frag_carros_context.xml

```
<menu . . .>
    <item android:id="@+id/action_share" ...
        app:actionProviderClass="android.support.v7.widget.ShareActionProvider" />
    <item android:id="@+id/action_remove" . . . />
</menu>
```

Ao compartilhar as informações, precisamos criar um objeto do tipo `android.content.Intent` e preencher com os dados necessários. Essa intent é configurada no `ShareActionProvider`. Com isso, o Android poderá disparar uma mensagem de broadcast para todas as aplicações instaladas no dispositivo, com a intenção de que cada uma delas responda se é capaz de tratar o compartilhamento ou não. Por isso, ao compartilhar fotos, você pode escolher entre enviá-las por Gmail, Hangouts, WhatsApp etc. Tendo em vista que nosso objetivo é compartilhar os carros, vamos atualizar o código conforme demonstrado a seguir.

 CarrosFragment.java

```
. . .
import android.support.v7.widget.ShareActionProvider;
public class CarrosFragment extends BaseFragment {
    public Intent shareIntent;
    . . .
    @Override
    public boolean onCreateActionMode(ActionMode mode, Menu menu) {
        // Infla o menu específico da action bar de contexto (CAB)
        MenuInflater inflater = getActivity().getMenuInflater();
        inflater.inflate(R.menu.menu_frag_carros_context, menu);
        MenuItem shareItem = menu.findItem(R.id.action_share);
        ShareActionProvider share = (ShareActionProvider) M
            enuItemCompat.getActionProvider(shareItem);
        shareIntent = new Intent(Intent.ACTION_SEND);
        shareIntent.setType("text/*");
        share.setShareIntent(shareIntent);
        return true;
    }
    . . .
    private void updateActionModeTitle() {
        if (actionMode != null) {
            // Atualiza o título da CAB aqui
```

```
        updateShareIntent(selectedCarros);
    }
}
// Atualiza a share intent com os carros selecionados
private void updateShareIntent(List<Carro> selectedCarros) {
    if(shareIntent != null) {
        // Texto com os carros
        shareIntent.putExtra(Intent.EXTRA_TEXT, "Carros: " + selectedCarros);
    }
}
}
```

Nota: no código-fonte, importe a classe `android.support.v7.widget.ShareActionProvider` e não a classe `android.widget.ShareActionProvider`, pois estamos utilizando a biblioteca de compatibilidade.

O segredo é criar a intent de compartilhamento no método `onCreateActionMode()` logo após inflar o menu com as ações da CAB. Observe que o objeto da intent é salvo no atributo `shareIntent`. Depois, sempre que o usuário selecionar algum carro, o método `updateActionModeTitle()` é chamado para atualizar o título da CAB, com o objetivo de mostrar o texto “1 carro selecionado” ou “x carros selecionados”. Neste momento, vamos chamar o método `updateShareIntent()` para configurar e atualizar o conteúdo da intent com uma string que tem o nome de todos os carros.

Dica: a classe `android.content.Intent` é o coração do Android e tudo gira em torno dela. Até o momento, tínhamos visto que uma intent é utilizada para fazer a navegação de telas com o método `startActivity(intent)`. Agora vimos que uma intent também é utilizada para compartilhar informações. Na verdade, uma intent representa uma mensagem da aplicação para o sistema operacional, solicitando que algo seja realizado. Cada aplicativo pode tratar essa ação se for de seu interesse. Para mais detalhes, verifique o capítulo 20 sobre a classe `Intent`.

Para testar a teoria, recomendo instalar a aplicação em um dispositivo real, pois ele tem mais aplicações para tratar o compartilhamento. No caso do emulador, conforme a figura 193, podemos enviar os carros selecionados por SMS. Veja que até o momento a intent de compartilhamento foi configurada para enviar apenas o nome dos carros em modo texto, mas uma futura melhoria seria compartilhar mais informações, como as fotos.

19.6 Compartilhando as fotos dos carros selecionados

Para compartilhar as fotos dos carros, precisamos salvá-las em arquivos, para na sequência criar uma intent com o conteúdo desses arquivos. Uma maneira de fazer isso é salvar a foto do carro em arquivo no mesmo instante em que mostramos a foto no adapter, mas desse jeito podemos gastar recursos demais, uma vez que todas as fotos seriam salvas sem necessidade.

Outra forma é, no momento em que o usuário clicar em compartilhar, disparar uma tarefa (Thread ou AsyncTask) para fazer o download das fotos para arquivo, para somente depois enviar a intent com o compartilhamento. Para o aplicativo dos carros, vou escolher essa segunda opção. No tópico anterior, pedi para você configurar a classe `android.support.v7.widget.ShareActionProvider` no arquivo de menu, mas desta vez vou pedir para você removê-la. Portanto, remova essa linha indicada do arquivo de menu.

 /res/menu/menu_frag_carros_context.xml

```
<menu . . >
    <item android:id="@+id/action_share" ...
        app:actionProviderClass="android.support.v7.widget.ShareActionProvider" />
    <item android:id="@+id/action_remove" . . . />
</menu>
```

O motivo por não utilizar a classe `ShareActionProvider` é porque, ao tocar no botão de compartilhar, vamos disparar uma tarefa para fazer o download das fotos. E caso a classe `ShareActionProvider` seja utilizada, ela toma conta de tudo, e não nos deixa tratar o evento. Portanto, apague tudo o que fizemos no tópico anterior, para deixar o código exatamente como estava antes de iniciar a parte de compartilhamento. Peço desculpas por isso, mas é que eu precisava explicar a classe `ShareActionProvider`, porém, para este exemplo mais avançado, eu prefiro não utilizá-la.

Você deve remover a parte do código dentro do método `onCreateActionMode()` que configura o `ShareActionProvider`, remova o método `updateShareIntent()` e também o atributo da classe `Intent shareIntent`. Quando o código estiver sem a função de compartilhamento, vamos continuar.

Na ação `R.id.action_share` no método `onOptionsItemSelected()`, vamos chamar uma tarefa para fazer o download das fotos para arquivo. Logo em seguida, vamos disparar uma intent que vai conter a lista dos arquivos para serem compartilhados. Para isso, deixe o código conforme demonstrado a seguir:

 CarrosFragment.java

```

public class CarrosFragment extends BaseFragment {
    . . .
    @Override
    public boolean onOptionsItemSelected(ActionMode mode, MenuItem item) {
        List<Carro> selectedCarros = getSelectedCarros();
        if (item.getItemId() == R.id.action_remove) {
            . . .
        } else if (item.getItemId() == R.id.action_share) {
            // Dispara a tarefa para fazer download das fotos
            startTask("compartilhar", new CompartilharTask(selectedCarros));
        }
        // Encerra o action mode
        mode.finish();
        return true;
    }
    . . .
    // Task para fazer o download
    // Faça import da classe android.net.Uri;
    private class CompartilharTask implements TaskListener {
        // Lista de arquivos para compartilhar
        ArrayList<Uri> imageUris = new ArrayList<Uri>();
        private final List<Carro> selectedCarros;
        public CompartilharTask(List<Carro> selectedCarros) {
            this.selectedCarros = selectedCarros;
        }
        @Override
        public Object execute() throws Exception {
            if(selectedCarros != null) {
                for (Carro c : selectedCarros) {
                    // Faz o download da foto do carro para arquivo
                    String url = c.urlFoto;
                    String fileName = url.substring(url.lastIndexOf("/"));
                    // Cria o arquivo no SD card
                    File file = SDCardUtils.getPrivateFile(getContext(), "carros", fileName);
                    IOUtils.downloadToFile(c.urlFoto, file);
                    // Salva a Uri para compartilhar a foto
                    imageUris.add(Uri.fromFile(file));
                }
            }
            return null;
        }
    }
}

```

```

@Override
public void updateView(Object o) {
    // Cria a intent com a foto dos carros
    Intent shareIntent = new Intent();
    shareIntent.setAction(Intent.ACTION_SEND);
    shareIntent.setAction(Intent.ACTION_SEND_MULTIPLE);
    shareIntent.putParcelableArrayListExtra(Intent.EXTRA_STREAM, imageUris);
    shareIntent.setType("image/*");
    // Cria o Intent Chooser com as opções
    startActivity(Intent.createChooser(shareIntent, "Enviar Carros"));
}
@Override
public void onError(Exception e) { alert("Ocorreu algum erro ao compartilhar."); }
@Override
public void onCancelled(String s) { }
}
}

```

Ao compartilhar os carros, é disparada uma tarefa/thread, da mesma forma que fizemos ao consultar os carros do web service. O método `execute()` vai executar em segundo plano, sendo responsável por fazer o download das fotos para arquivo e também por salvar uma lista do tipo `ArrayList<Uri>` com esses arquivos. A classe `android.net.Uri` representa um recurso acessado por um caminho. Na prática, a `Uri` contém a localização do arquivo. O método `updateView()` é executado na `UIThread` quando o download das fotos terminar; neste momento, estamos criando a intent, e configurando-a com as fotos que precisam ser enviadas.

Algo interessante deste exemplo é o método `Intent.createChooser(intent,titulo)`, que cria um alerta para o usuário escolher com qual aplicação ele deseja compartilhar as informações. Segundo a documentação do Android, utilizar este escolhedor de aplicativos (chooser) tem três vantagens:

1. O alerta com as opções será exibido sempre, mesmo se o usuário já selecionou algum aplicativo para ser o compartilhador padrão.
2. Se nenhuma aplicação conseguir tratar a mensagem enviada pela intent, será mostrado um alerta com uma mensagem de erro padrão. Caso você não utilize o chooser, ocorre um erro em tempo de execução.
3. Podemos definir um título para a janela do alerta.

A figura 194 mostra o resultado deste exemplo. Ao selecionar alguns carros e clicar no botão compartilhar, aparece o alerta com as aplicações que podem compartilhar as fotos.

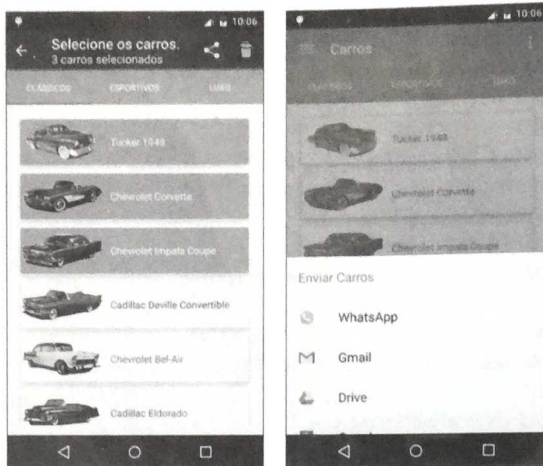


Figura 19.4 – Compartilhando fotos.

A figura 19.5 mostra o resultado ao escolher o aplicativo do Gmail para compartilhar as fotos.

Dica: para testar melhor a funcionalidade de compartilhamento, utilize um dispositivo real.

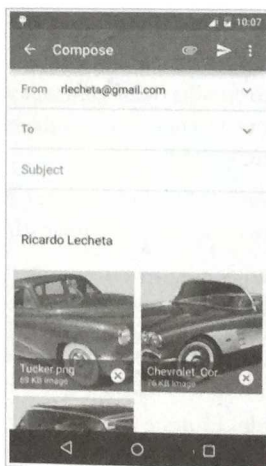


Figura 19.5 – Compartilhando fotos pelo Gmail.

19.7 Links úteis

Neste capítulo, estudamos como habilitar a action bar de contexto (CAB) para selecionar múltiplas linhas de uma lista e também fizemos um exemplo para compartilhar informações. Para continuar seus estudos, separei alguns links da documentação oficial.

- **Android API Guides – Contextual Menus**

<http://developer.android.com/guide/topics/ui/menus.html>

- **Android API Guides – Contextual Action Mode (CAB)**

<http://developer.android.com/guide/topics/ui/menus.html#CAB>

- **Android Training – Sharing Simple Data**

<http://developer.android.com/training/sharing/index.html>



CAPÍTULO 20

Intents

Neste capítulo, vamos fazer uma pausa no desenvolvimento do projeto dos carros para aprender detalhes importantes sobre a classe `android.content.Intent`, que é o coração do Android. Uma intent é uma mensagem da aplicação para o sistema operacional, solicitando que algo seja realizado, e representa um importante papel na arquitetura do Android para integrar diferentes aplicações.

Cabe ao sistema operacional interpretar essa mensagem e tomar as providências necessárias, que pode ser abrir um aplicativo, fazer uma ligação para determinado número, tirar uma foto ou até abrir o browser em uma página da internet.

Demonstraremos diversos exemplos de como utilizar a classe `Intent` e aprenderemos a chamar algumas aplicações nativas do Android.

20.1 Intent – envio de uma mensagem ao Android

A classe `android.content.Intent` representa uma ação que a aplicação deseja executar. A tradução de intent para o português é intenção, de forma que podemos dizer que uma intent representa a intenção da aplicação de realizar determinada tarefa. Na prática, essa intenção é enviada ao sistema operacional como uma mensagem, chamada de broadcast. Ao receber a mensagem, o sistema operacional tomará as decisões necessárias, dependendo do conteúdo da mensagem.

Uma intent pode ser utilizada para:

- enviar uma mensagem para o sistema operacional;
- abrir uma nova tela da aplicação, utilizando o método `startActivity(intent)`;
- ligar para um número de celular;
- abrir o browser em uma página da internet;
- enviar uma mensagem por SMS ou email;

- exibir algum endereço, localização ou rota no Google Maps;
- abrir a agenda de contatos para selecionar um contato;
- abrir a galeria de fotos ou vídeos para selecionar um arquivo de multimídia;
- tocar uma música ou vídeo;
- abrir a câmera e solicitar que uma foto ou vídeo sejam gravados;
- enviar mensagens de uma tela para outra dentro do mesmo aplicativo, ou até trocar mensagens entre diferentes aplicativos;
- integrar diferentes aplicações; por exemplo, é possível enviar uma mensagem para aplicativos especializados em ler um código de barras para obter a resposta;
- abrir o Google Play para fazer a instalação de determinado aplicativo;
- executar algum processamento pesado em segundo plano usando as classes `BroadcastReceiver` e `Service`, que serão explicadas em outros capítulos;
- e muito mais.

Nota: uma intent é entre muitas coisas a forma utilizada para fazer com que aplicações em processos diferentes se comuniquem, utilizando uma mensagem que, dependendo do seu conteúdo, pode ser interceptada por qualquer aplicação.

20.2 Intents explícitas e implícitas

Nesta altura do campeonato, você já sabe navegar entre as telas da aplicação, e para isso utilizamos uma intent. Neste tópico, vamos revisar alguns conceitos e aprender a diferença entre intents explícitas e implícitas.

Uma activity representa uma tela da aplicação, e para navegar entre essas telas utilizamos o método `startActivity(intent)`. O parâmetro `intent` é uma instância da classe `android.content.Intent` que contém as informações necessárias referentes à activity que será executada. Por exemplo:

```
// Intent explícita
Intent intent = new Intent(getContext(), CarroActivity.class);
intent.putExtra("carro", carro);
startActivity(intent);
```

Esse trecho de código na verdade criou uma mensagem para o sistema operacional, cujo conteúdo é: “Minha intenção/vontade é abrir a tela do carro”. Ao chamar o método `startActivity(intent)`, essa mensagem é enviada ao sistema operacional.

Cabe ao sistema operacional analisar o conteúdo da mensagem e encontrar a tela correta que precisa ser aberta. A intent deste exemplo de navegação para a tela do carro é chamada de **intent explícita**, pois ela especifica exatamente qual activity deve ser chamada. Isso é feito quando você precisa abrir uma activity que faz parte do seu próprio aplicativo. Mas lembre-se: uma intent é muito mais do que isso, pois na prática é uma mensagem enviada ao sistema operacional.

O interessante de uma mensagem enviada por uma intent é que nem sempre ela é recebida pela própria aplicação que a criou, por isso ela é frequentemente utilizada para integrar aplicações. Para exemplificar, veja este exemplo do método `startActivity(intent)`, que utiliza a ação `ACTION_CALL` e uma Uri em vez de especificar a activity:

```
// Intent implícita
Uri uri = Uri.parse("tel:12345678");
Intent intent = new Intent(Intent.ACTION_CALL, uri);
startActivity(intent);
```

Nesse caso, o método `startActivity(intent)` foi utilizado novamente, mas desta vez nenhuma activity foi especificada. Essa é uma **intent implícita**, que na prática representa uma mensagem enviada para todas as aplicações instaladas no sistema. Neste momento, caberá a cada aplicação interpretar a mensagem, informando ao Android quem poderá tratá-la.

Especificamente para esta mensagem "tel:12345678", quem vai tratá-la é a aplicação que faz ligações, portanto o resultado é que a activity da aplicação de ligação vai abrir e ligar para esse número. Esse exemplo inicial foi apenas para você entender que uma intent é utilizada para tudo, não apenas para navegar entre as activities do aplicativo.

Nota: uma intent **explícita** determina exatamente qual activity deve executar. Esse tipo de intent é utilizado quando a activity ou mensagem deve ser enviada para alguma classe que faz parte do seu aplicativo. Uma intent **implícita** é uma mensagem genérica enviada ao sistema operacional e pode ser tratada por qualquer aplicação instalada.

20.3 Exemplos de intents nativas

Acredito que a melhor maneira de entender o que é uma intent é partirmos para a prática; portanto, abra o projeto de exemplo deste capítulo no Android Studio. Seguindo o mesmo template de outros projetos do livro, a activity inicial do projeto contém um `ListView` com várias opções. Cada opção é um exemplo de intent que pode ser enviada ao sistema operacional.



MainActivity.java

```
package br.com.livroandroid.intents;

public class MainActivity extends AppCompatActivity implements
    AdapterView.OnItemClickListener {
    private static final String TAG = "livroandroid";
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        String[] items = new String[]{
            "Ligar para telefone", "Discar para telefone",
            "Enviar Email", "Enviar SMS",
            "Abrir Browser", "Mapa - Lat/Lng",
            "Mapa - Endereco", "Mapa - Rota",
            "Compartilhar", "Camera Foto",
            "Camera Video", "Visualizar Todos Contatos",
            "Visualizar Contato 1", "Selecionar Contato",
            "Intent customizada", "Intent customizada / schema",
            "Sair"
        };
        ListView listView = new ListView(this);
        setContentView(listView);
        listView.setOnItemClickListener(this);
        listView.setAdapter(new ArrayAdapter<String>(this,
            android.R.layout.simple_list_item_1, items));
    }
    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
        try {
            switch (position) {
                // Ligar para um número de telefone
                case 0:
                    Uri uri = Uri.parse("tel:12345678");
                    Intent intent = new Intent(Intent.ACTION_CALL, uri);
                    startActivity(intent);
                    break;
                // Discar para um número de telefone
                case 1:
                    uri = Uri.parse("tel:12345678");
                    intent = new Intent(Intent.ACTION_DIAL, uri);
                    startActivity(intent);
                    break;
                // Enviar um email
```

```

case 2:
    // Email
    Intent emailIntent = new Intent(Intent.ACTION_SEND);
    emailIntent.putExtra(Intent.EXTRA_SUBJECT, "Título do email");
    emailIntent.putExtra(Intent.EXTRA_TEXT, "Olá");
    emailIntent.putExtra(Intent.EXTRA_EMAIL, "rlecheta@gmail.com");
    emailIntent.setType("message/rfc822");
    startActivity(emailIntent);
    break;
// Enviar um SMS
case 3:
    // SMS
    uri = Uri.parse("sms:12345678");
    Intent smsIntent = new Intent(Intent.ACTION_SENDTO, uri);
    smsIntent.putExtra("sms_body", "Olá");
    startActivity(smsIntent);
    break;
// Abrir o browser em uma página
case 4:
    // Browser
    uri = Uri.parse("http://google.com");
    intent = new Intent(Intent.ACTION_VIEW, uri);
    startActivity(intent);
    break;
// Abrir o mapa na coordenada: Latitude/Longitude
case 5:
    // Mapa
    String GEO_URI = "geo:-25.4089185,-49.3222402";
    intent = new Intent(Intent.ACTION_VIEW, Uri.parse(GEO_URI));
    startActivity(intent);
    break;
// Abrir o mapa em um endereço
case 6:
    // Mapa
    GEO_URI = "geo:0,0?q=Av. Manoel Ribas - Santa Felicidade, Curitiba - Paraná, Brasil";
    intent = new Intent(Intent.ACTION_VIEW, Uri.parse(GEO_URI));
    startActivity(intent);
    break;
// Abrir o mapa e mostrar a rota entre dois endereços
case 7:
    String rota =

```

```
"http://maps.google.com/maps?saddr=-25.4089185,-49.3222402&dad
dr=-25.428781,-49.30925";
    intent = new Intent(Intent.ACTION_VIEW, Uri.parse(rota));
    startActivity(intent);
    break;
// Compartilhar informações
case 8:
    Intent shareIntent = new Intent(Intent.ACTION_SEND);
    shareIntent.setType("text/plain");
    shareIntent.putExtra(Intent.EXTRA_SUBJECT, "Compartilhar");
    shareIntent.putExtra(Intent.EXTRA_TEXT, "Bla bla bla");
    startActivity(shareIntent);
    break;
// Abrir a câmera para tirar uma foto
case 9:
    Intent fotoIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    startActivityForResult(fotoIntent, 9);
    break;
// Abrir a câmera para gravar um vídeo
case 10:
    Intent videoIntent = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
    startActivityForResult(videoIntent, 0);
    break;
// Abrir a agenda para visualizar todos os contatos
case 11:
    uri = Uri.parse("content://com.android.contacts/contacts");
    intent = new Intent(Intent.ACTION_VIEW, uri);
    startActivity(intent);
    break;
// Abrir a agenda para visualizar um contato
case 12:
    uri = Uri.parse("content://com.android.contacts/contacts/1");
    intent = new Intent(Intent.ACTION_VIEW, uri);
    startActivity(intent);
    break;
// Abrir a agenda para escolher um contato
case 13:
    uri = Uri.parse("content://com.android.contacts/contacts");
    intent = new Intent(Intent.ACTION_PICK, uri);
    startActivityForResult(intent, 13);
    break;
// Intent com ação customizada
case 14:
```

```

        intent = new Intent("br.com.livroandroid.intents.TESTE");
        startActivity(intent);
        break;
    // Intent com ação customizada especificando um 'schema'
    case 15:
        // INTENT_FILTER
        uri = Uri.parse("livroandroid://carros/ferrari");
        intent = new Intent(Intent.ACTION_VIEW, uri);
        startActivity(intent);
        break;
    default:
        finish();
        break;
    }
} catch (Exception e) {
    Toast.makeText(this, "Erro :" + e.getMessage(), Toast.LENGTH_SHORT).show();
}
}
}
@Override
protected void onActivityResult(int codigo, int resultado, Intent it) {
    Log.i(TAG, "Menu.onActivityResult: " + codigo + ", resultado: " + resultado + " > " + it);
    if (codigo == 9 && resultado == Activity.RESULT_OK) {
        // Tirou a foto, aqui podemos ler o Bitmap
    }
    else if (codigo == 13 && resultado == Activity.RESULT_OK) {
        // Selecionou o contato
        Uri uri = it.getData();
        Log.d(TAG, "Contato Uri: " + uri);
    }
}
}
}
}

```

Neste código, temos vários exemplos de intents implícitas que serão enviadas como mensagem para o sistema operacional. Cada intent tem como objetivo realizar alguma ação, como abrir o browser, abrir o mapa, ligar para um telefone etc. Recomendo que você execute este projeto no emulador ou em um dispositivo para conferir o resultado. Veja que o código-fonte está comentado, portanto leia com calma.

Nota: algumas destas intents vão acessar a agenda de contatos, adicione alguns contatos no emulador para testar. Note que existe uma intent no código que tem como objetivo mostrar o contato de id=1, então naturalmente esse contato precisa existir.

Veja que existem muitas maneiras de criar uma intent. Um dos construtores da classe `android.content.Intent` aceita apenas uma string que pode ser qualquer coisa, como “abacaxi” ou “bingo”. Essa string é a ação que a mensagem vai executar. Por convenção, a ação é composta do nome do pacote, mais um código, como por exemplo:

```
// Intent implícita: Que vai tratá-la?
Intent intent = new Intent("br.com.livroandroid.intents.TESTE");
startActivity(intent);
```

Como essa é uma intent implícita que tem uma ação genérica, alguma aplicação deve entender o que significa essa ação. Depois vamos estudar mais detalhes sobre isso, mas, apenas para adiantar o assunto, essa intent vai abrir a `activity TesteActivity`. Isso acontece porque no manifesto da aplicação foi configurado um filtro de intenção, conforme mostra o trecho de código a seguir:

```
// AndroidManifest.xml
<activity android:name=".TesteActivity" . . .>
    <intent-filter>
        <action android:name="br.com.livroandroid.intents.TESTE" />
        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>
</activity>
```

Outro construtor disponível na classe `android.content.Intent` aceita um parâmetro que é a ação e uma `Uri` com o conteúdo; por exemplo:

```
Uri uri = Uri.parse("tel:12345678");
Intent intent = new Intent(Intent.ACTION_CALL, uri);
startActivity(intent);
```

Existem muitas ações padrões da plataforma, e cada uma delas apresenta constantes na classe `Intent`. Na prática, cada ação é uma `String`, mas utilizam-se as constantes para facilitar o código. Algumas das ações mais comuns são `Intent.ACTION_VIEW`, `Intent.ACTION_CALL`, `Intent.ACTION_DIAL`, `Intent.ACTION_SEND`, `Intent.ACTION_IMAGE_CAPTURE` etc. No código anterior, a ação é `Intent.ACTION_CALL`; por isso, quando a aplicação abrir, ela vai entender que deve efetuar a ligação telefônica. Às vezes a mesma aplicação pode receber diferentes tipos de mensagens, portanto se trocar a ação para `Intent.ACTION_DIAL` a aplicação do telefone vai abrir novamente, mas desta vez vai apenas discar o número, sem efetuar a ligação. Na prática, cada aplicação nativa também é uma `activity` e cada `activity` pode decidir o que fazer com base na ação e parâmetros recebidos pela intent. A figura 20.1 mostra o projeto de exemplo deste capítulo executando no emulador, e o primeiro exemplo que faz a ligação para um número de telefone.

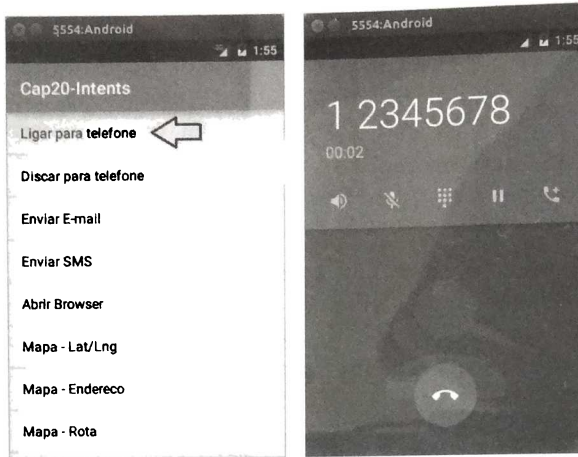


Figura 20.1 – Projeto exemplo com intents.

20.4 Permissões

Algumas intents precisam de permissões para funcionar, como nos casos de fazer ligação, listar os contatos da agenda e abrir o browser. Portanto, este breve tópico serve apenas para alertá-lo de que essas permissões foram configuradas no projeto de exemplo deste capítulo.

```
<uses-permission android:name="android.permission.CALL_PHONE" />
<uses-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.INTERNET" />
```

20.5 Retornando resultados de uma intent – startActivityForResult

Se você observar nos exemplos, as intents da câmera e da agenda de contatos foram disparadas com o método `startActivityForResult(intent, código)`, pois estamos interessados na resposta. No caso da aplicação da câmera, a resposta será a foto; e no caso da aplicação da agenda, a resposta é o contato selecionado.

Fique tranquilo se você achou os exemplos sobre câmera e contatos simples demais, pois este capítulo é apenas uma breve introdução sobre as intents. Posteriormente, teremos capítulos dedicados aos recursos de multimídia e agenda de contatos. Sobre o exemplo que tem como objetivo selecionar um contato, é utilizada uma `Uri`

customizada do provedor de conteúdo dos contatos e a ação `Intent.ACTION_PICK`, que por convenção indica que a intent deve retornar algum resultado.

```
// Abre a agenda para selecionar um contato
Uri uri = Uri.parse("content://com.android.contacts/contacts");
Intent intent = new Intent(Intent.ACTION_PICK, uri);
startActivityForResult(intent, 13);
```

O método `startActivityForResult(intent,codigo)` recebe dois parâmetros:

Parâmetro	Descrição
<code>intent</code>	Intent que representa a mensagem.
<code>int codigo</code>	Código numérico utilizado para identificar a activity que será iniciada, chamado de <code>requestCode</code> no Android. O mesmo valor será passado ao método <code>onActivityResult(codigo, resultado, intent)</code> quando a activity chamada finalizar, portanto este código pode ajudar a identificar o retorno de cada chamada.

Neste exemplo, a agenda de contatos vai abrir, logo cadastre alguns contatos no emulador para testar. Depois que você selecionar algum contato, ele será enviado como resultado para a activity de origem, ou seja, a `MainActivity` no nosso caso. O resultado é enviado para o método `onActivityResult(...)` que deve ser sobrescrito na activity que fez a chamada da intent. No trecho de código citado, o código=13 foi o utilizado para chamar a activity dos contatos. Nesse contexto estamos testando se o retorno é dessa activity e se o resultado é `RESULT_OK`.

```
protected void onActivityResult(int codigo, int resultado, Intent it) {
    . . .
    if (codigo == 13 && resultado == Activity.RESULT_OK) {
        // A Uri representa o endereço do contato selecionado
        Uri uri = it.getData();
        Log.d(TAG, "Contato Uri: " + uri);
    }
}
```

Cada aplicação retorna o resultado de uma maneira diferente, mas no caso da agenda o retorno é enviado pela Uri que representa o contato selecionado. Por meio dessa Uri, podemos ler as informações do contato, como nome, email, telefone, foto etc. Mas isso vamos deixar para outro capítulo.

O método `onActivityResult(codigo,resultado,intent)` recebe os parâmetros.

Parâmetro	Descrição
<code>int codigo</code>	Mesmo código que originou a chamada com o método <code>startActivityForResult(intent, codigo)</code> .

Parâmetro	Descrição (cont.)
int resultado	Constante que indica se o resultado foi bem-sucedido ou não. Pode ser qualquer constante definida na aplicação. Por padrão, utilizam-se as constantes <code>Activity.RESULT_OK</code> e <code>RESULT_CANCELED</code>
intent	Intent que originou o retorno. Com essa intent é possível recuperar o Bundle para ler os parâmetros retornados. O método <code>intent.getData()</code> retorna uma Uri, que algumas aplicações utilizam para enviar a resposta.

20.6 IntentFilter

Quando uma mensagem é enviada ao sistema operacional utilizando a classe Intent, é necessário configurar a classe `IntentFilter` para interceptar essa mensagem, com base em seu conteúdo. Por exemplo, uma das intents que fazem parte do projeto de exemplo deste capítulo é esta:

```
// Este código abre a TesteActivity, mas por quê?
Intent intent = new Intent("br.com.livroandroid.intents.TESTE");
startActivity(intent);
```

Ao enviar essa mensagem, o Android abrirá a activity que foi configurada para interceptar a ação. Essa configuração deve ser feita com a tag `<intent-filter>` no arquivo de manifesto. A seguir podemos visualizar o código do arquivo *AndroidManifest.xml* do projeto deste capítulo. Veja que a activity `TesteActivity` foi configurada para interceptar exatamente a ação `br.com.livroandroid.intents.TESTE`.



AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest . . .>
  <application . . . >
    <activity android:name=".MainActivity" . . .>
      <activity android:name=".TesteActivity" . . .>
        <intent-filter>
          <action android:name="br.com.livroandroid.intents.TESTE" />
          <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>
      <intent-filter>
        <action android:name="android.intent.action.VIEW" />
        <category android:name="android.intent.category.DEFAULT" />
        <data android:scheme="livroandroid" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Por isso, ao disparar a intent com a ação "br.com.livroandroid.intents.TESTE", a classe TesteActivity é executada. A tag <intent-filter> corresponde à classe android.content.IntentFilter, que como você já deve ter entendido é utilizada para mapear uma ação para determinada classe. Seguindo o mesmo raciocínio, se a mensagem "bingo" for enviada, tal como no exemplo a seguir:

```
Intent intent = new Intent("bingo");
startActivity(intent);
```

Basta declarar o seguinte filtro em qualquer activity para receber essa mensagem.

```
<intent-filter>
    <action android:name="bingo" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

Nota: apenas para reforçar o conceito, note que nunca declaramos um filtro para intents explícitas, pois elas especificam exatamente qual activity deve abrir. Filtros são utilizados apenas para mapear ações genéricas, no caso de intents implícitas.

A grande mágica e o pulo do gato desta arquitetura de mensagens é que a dupla Intent e IntentFilter está presente em todos os lugares. Por exemplo, quando o sistema operacional recebe uma mensagem SMS, ele dispara uma intent com a ação android.provider.Telephony.SMS_RECEIVED, que pode ser configurada por qualquer aplicação que deseja interceptar essa intent, para, por exemplo, ler a mensagem SMS. No capítulo 33, sobre SMS, vamos aprender a fazer isso.

O sistema operacional do Android dispara essas intents para tudo. Por exemplo, se o Wi-Fi foi ligado ou desligado, podemos interceptar essas mensagens a fim de executar determinada lógica na aplicação. Aos poucos você vai entendendo esses conceitos e aprendendo a filtrar essas mensagens; então fique tranquilo, pois o objetivo principal deste capítulo foi ensiná-lo sobre essa arquitetura e a grande importância de uma intent.

Nota: se traduzirmos literalmente a expressão IntentFilter, teríamos a tradução "filtro de intenções". Nesse caso, vamos dizer que o IntentFilter nada mais é do que um filtro de uma intent (intenção). Na prática, é esse filtro quem decide se a mensagem daquela intent lhe interessa ou não, e, em caso afirmativo, a activity é executada.

20.7 Por que a MainActivity declara um <intent-filter>?

Sempre que criamos um novo projeto, automaticamente o wizard cria a activity principal, que por convenção tem o nome MainActivity. Essa activity é configurada automaticamente com um <intent-filter>, conforme demonstrado a seguir:

```
<activity android:name=".MainActivity" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

No caso da MainActivity, são declaradas na maioria das vezes a ação `android.intent.action.MAIN` e a categoria `android.intent.category.LAUNCHER`. Isso é feito para adicionar o ícone da aplicação na tela inicial do Android, para o usuário clicar no ícone e abrir a aplicação.

Para revisar o conceito, temos a seguir a definição da ação MAIN e categoria LAUNCHER:

- `android.intent.action.MAIN` – Ação-padrão do Android, usada para abrir uma activity como o ponto de partida, sem depender de outra activity. Se uma activity está configurada com a ação MAIN, ela pode ser iniciada diretamente pelo usuário ou Android Studio; caso contrário, será executada somente a partir do método `startActivity(intent)`. É como definir o famoso método `public static void main(String args[])` em uma aplicação Java.
- `android.intent.category.LAUNCHER` – Categoria que faz com que esta activity fique presente na tela inicial do Android, ou seja, esta activity é o ponto de partida da aplicação e pode ser iniciada pelo usuário.

20.8 Exemplo completo com intent customizada

Agora que já estudamos o básico sobre o que é uma mensagem enviada pela intent e aprendemos a interceptar essas mensagens com um intent filter, vamos criar um exemplo customizado de intent para abrir a lista de carros.

O exemplo que vamos criar é parecido com a intent que mostra toda a lista de contatos e permite selecionar um contato, de forma que o contato selecionado é retornado para a aplicação. Apenas para lembrar, essa intent dos contatos pode ser disparada com este código:

```
// Abre a agenda para selecionar um contato
Uri uri = Uri.parse("content://com.android.contacts/contacts");
Intent intent = new Intent(Intent.ACTION_PICK, uri);
startActivityForResult(intent, 13);
```

O resultado já sabemos que será entregue no método `onActivityResult(...)`. O nosso objetivo será fazer a mesma coisa, mas vamos disparar uma intent que vai mostrar a lista de carros, e ao selecionar um carro ele será retornado para a activity que fez a chamada. A seguir podemos ver um exemplo desta intent.

```
// Abre a lista de carros para selecionar um carro
Uri uri = Uri.parse("carros://br.com.livroandroid.carros/carros");
Intent intent = new Intent(Intent.ACTION_PICK, uri);
startActivityForResult(intent, 99);
```

Da mesma forma que podemos selecionar apenas um contato com a ação `content://com.android.contacts/contacts/1`, podemos criar uma activity que mostre o carro pela ação `carros://br.com.livroandroid.carros/carros/1`, na qual o número 1 é o identificador do registro. Porém para este exemplo vou preferir utilizar uma ação que tem o nome do carro, assim não corremos risco de buscar um carro que não existe, caso o id não seja encontrado.

Vamos então criar uma intent que possa abrir um carro pelo nome.

```
// Abre o carro pelo nome
Uri uri = Uri.parse("carros://br.com.livroandroid.carros/carros/Ferrari FF");
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

Antes de começarmos, vale ressaltar que esse tipo de intent é muito específica, e dificilmente você vai precisar fazer este tipo de coisa no seu dia a dia. Ao desenvolver aplicativos, geralmente você navega entre as activities utilizando intent explícitas. Estamos estudando isso porque quero que você entenda como o Android funciona por debaixo dos panos.

Outro ponto que você precisa entender é que estas mensagens podem ser disparadas de qualquer aplicação instalada. No projeto dos carros, vamos configurar um filtro de intenção (`intent-filter`) para interceptar essas mensagens. Portanto vamos abrir uma porta na aplicação dos carros, para que ela possa fornecer conteúdo, de modo a comunicar-se com outras aplicações. Para continuar, faça uma cópia do projeto dos carros e crie a seguinte activity:



CarrosIntentActivity.java

```
package br.com.livroandroid.carros.activity;

...

public class CarrosIntentActivity extends BaseActivity {
    private List<Carro> carros;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_carros_intent);
        // (*1*) Lê as informações da intent
        Intent intent = getIntent();
        Uri uri = intent.getData();
        Log.d("livroandroid", "Action: " + intent.getAction());
        Log.d("livroandroid", "Scheme: " + uri.getScheme());
        Log.d("livroandroid", "Host: " + uri.getHost());
        Log.d("livroandroid", "Path: " + uri.getPath());
        Log.d("livroandroid", "PathSegments: " + uri.getPathSegments());
        // RecyclerView
        RecyclerView recyclerView = (RecyclerView) findViewById(R.id.recyclerView);
        LinearLayoutManager layoutManager = new LinearLayoutManager(this);
        recyclerView.setLayoutManager(layoutManager);
        // Configura a Toolbar como a action bar
        setUpToolbar();
        CarroDB db = new CarroDB(this);
        try {
            if("/carros".equals(uri.getPath())) {
                // (*2*) Lista todos os carros
                this.carros = db.findAll();
                recyclerView.setAdapter(new CarroAdapter(this, carros, onClickCarro()));
            } else {
                // (*3*) Busca o carro pelo nome: /carros/Ferrari FF
                List<String> paths = uri.getPathSegments();
                String nome = paths.get(1);
                Carro carro = db.findByNome(nome);
                if(carro != null) {
                    // Se encontrou o carro, abre a activity para mostrá-lo
                    Intent carroIntent = new Intent(this, CarroActivity.class);
                    carroIntent.putExtra("carro", carro);
                    startActivity(carroIntent);
                    finish();
                }
            }
        }
    }
}
```

```

        } finally {
            db.close();
        }
    }

    private CarroAdapter.CarroOnClickListener onClickCarro() {
        return new CarroAdapter.CarroOnClickListener() {
            @Override
            public void onClickCarro(View view, int idx) {
                Carro c = carros.get(idx);
                // (*4*) Retorna o carro selecionado para quem chamou
                Intent intent = new Intent();
                intent.putExtra("nome",c.nome);
                intent.putExtra("url_foto",c.urlFoto);
                setResult(Activity.RESULT_OK,intent);
                finish();
            }
            @Override
            public void onLongClickCarro(View view, int idx) {
                // nada aqui
            }
        };
    }
}

```

Essa activity busca todos os carros que estão salvos no banco de dados, e para simplificar o código note que não utilizei nenhuma thread (AsyncTask), mas na prática isso seria necessário. Veja que adicionei algumas anotações no código, explicadas logo a seguir.

Na parte (*1*) estamos extraindo as informações da intent e de sua Uri. Com base nessas informações, podemos decidir o que fazer.

```

Log.d("livroandroid", "Action: " + intent.getAction()); // Imprime ACTION_VIEW ou ACTION_PICK
Log.d("livroandroid", "Scheme: " + uri.getScheme()); // Imprime carro://
Log.d("livroandroid", "Host: " + uri.getHost()); // Imprime br.com.livroandroid
Log.d("livroandroid", "Path: " + uri.getPath()); // Imprime /carros ou /carros/nome

```

Isso é feito para extrair as partes da Uri utilizada pela intent, que é `carros://br.com.livroandroid.carros/carros` ou `carros://br.com.livroandroid.carros/carros/nome_carro`.

A parte (*2*) é o código que mostra todos os carros em uma lista com o RecyclerView, e isso é feito se o método `uri.getPath()` é igual a `/carros`. A parte (*3*) é o código que lê o nome do carro, extraindo do path da Uri. Neste caso, o carro é buscado no banco de dados pelo nome, e se for encontrado a activity de detalhes do carro

é chamada. A parte (*4*) é o código que trata o evento de seleção da lista, neste caso é criada uma intent de resultado, com os parâmetros que são o nome e a foto do carro. O método `setResult(código,intent)` é utilizado para enviar a intent de retorno. Sendo assim, quando o método `finish()` for chamado, a activity que fez a chamada da intent receberá o resultado no método `onActivityResult(...)`.

Estas explicações são sucintas, pois espero que você dê uma boa estudada no código para melhorar seu aprendizado. Vamos continuar com a configuração. Como essa activity precisa abrir ao receber determinada mensagem, configure o arquivo de manifesto conforme demonstrado a seguir:

AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest . . . >
    <application . . . >
        . . .
        <activity android:name=".activity.CarrosIntentActivity" . . . >
            <!-- Filtro para carros://br.com.livroandroid.carros/carros/* -->
            <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:scheme="carros" />
            </intent-filter>
            <!-- Filtro para carros://br.com.livroandroid.carros/carros/* -->
            <intent-filter>
                <action android:name="android.intent.action.PICK" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:scheme="carros" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

O primeiro filtro é para tratar a ação `Intent.ACTION_VIEW`, e o segundo para tratar a ação `Intent.ACTION_PICK`. Ambos os filtros declaram a tag `<data android:scheme="carros"/>` que corresponde à parte `carros://` da uri, chamada de **schema**. Da mesma forma que você poderia utilizar `sms://`, `tel://`, `http://` etc., se estivesse interessado em tratar outros tipos de mensagens. Portanto, nesses filtros você declara a ação que precisa ser interceptada e o **schema** que você está interessado. O que você coloca depois do **schema** não interessa, pois o restante da uri é como se fossem parâmetros que você pode ler no código.

Feito isso, execute novamente o projeto dos carros no emulador do Android, e o resultado é que esta nova activity será instalada na aplicação. Na sequência, feche o aplicativo dos carros.

Para testar a intent, vamos precisar criar um novo projeto que dispare estas duas intents. Assim, crie o projeto `Teste-Intent-Carros`. Este projeto contém apenas a `MainActivity` com um layout que tem dois botões para disparar as intents. No layout também temos um campo de texto para mostrar o nome do carro selecionado e uma imagem para mostrar a foto.



/res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="16dp" android:orientation="vertical">
    <TextView
        android:text="@string/hello_world"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <Button
        android:onClick="onClickMostrarCarroPeloNome"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Mostrar carro pelo nome" />
    <Button
        android:onClick="onClickSelecionarCarro"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Selecionar um carro" />
    <!-- Nome do carro -->
    <EditText
        android:id="@+id/tNomeCarro"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/nome_carro_selecionado"/>
    <!-- Foto do carro -->
    <ImageView
        android:id="@+id/ingFotoCarro"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:layout_margin="10dp"/>
</LinearLayout>
```

A seguir, podemos ver o código da `MainActivity`:



MainActivity.java

```

...
import com.squareup.picasso.Picasso;
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void onClickMostrarCarroPeloNome(View view) {
        Uri uri = Uri.parse("carros://br.com.livroandroid.carros/carros/Ferrari FF");
        Intent intent = new Intent(Intent.ACTION_VIEW, uri);
        startActivity(intent);
    }
    public void onClickSelecionarCarro(View view) {
        Uri uri = Uri.parse("carros://br.com.livroandroid.carros/carros");
        Intent intent = new Intent(Intent.ACTION_PICK, uri);
        startActivityForResult(intent, 99);
    }
    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        if(resultCode == Activity.RESULT_OK && requestCode == 99) {
            // Lê as informações do carro selecionado
            String nome = data.getStringExtra("nome");
            String url_foto = data.getStringExtra("url_foto");
            Log.d("livroandroid", "Foto: " + url_foto);
            // Mostra os dados do carro selecionado
            EditText text = (EditText) findViewById(R.id.tNomeCarro);
            ImageView img = (ImageView) findViewById(R.id.imgFotoCarro);
            text.setText(nome);
            Picasso.with(this).load(url_foto).into(img);
        }
    }
}

```

Para mostrar a foto do carro, a activity utiliza a biblioteca Picasso; logo, adicione essa dependência no arquivo *app/build.gradle* conforme fizemos no projeto dos carros. Como é preciso acessar a internet para mostrar a foto do carro, configure a permissão no arquivo *AndroidManifest.xml*.



AndroidManifest.xml

```
<manifest ... />
    <uses-permission android:name="android.permission.INTERNET" />
    <application ... />
</manifest>
```

Ao executar esse projeto no emulador, você verá uma tela com dois botões. Ao clicar no botão **Mostrar Carro pelo Nome**, a ação `ACTION_VIEW` com a Uri `"carros://br.com.livroandroid.carros/carros/Ferrari FF"` será disparada. O resultado será como a figura 20.2. Veja que interessante! Uma aplicação enviou uma mensagem para outra!

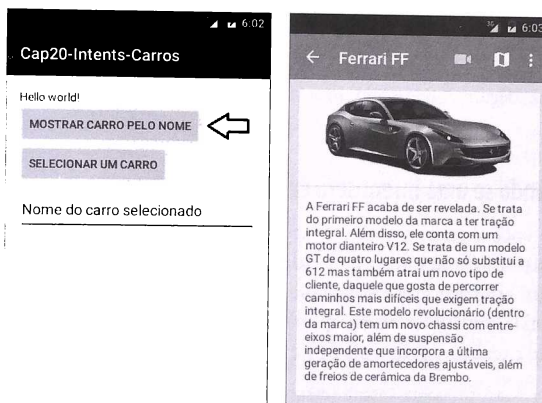


Figura 20.2 – Intent customizada para abrir um carro.

E mais interessante ainda é o segundo exemplo, que dispara a intent com a ação `ACTION_PICK` com a Uri `"carros://br.com.livroandroid.carros/carros"`. Neste caso, a aplicação dos carros mostra a lista para o usuário selecionar um carro. O carro selecionado é enviado como retorno para a activity, conforme mostra o fluxo da figura 20.3.

Esse exemplo demonstrou como integrar diferentes aplicações, e fizemos a mesma coisa que alguns aplicativos nativos fazem; portanto, agora você sabe como tudo funciona internamente.

Nota: qual método utilizar: `startActivity` ou `startActivityForResult`? Na maioria das vezes vamos utilizar o método `startActivity` para navegar entre as telas da aplicação, pois geralmente não estamos interessados em ler um retorno. Porém, em casos específicos nos quais é preciso obter um retorno da activity, devemos utilizar o método `startActivityForResult`.

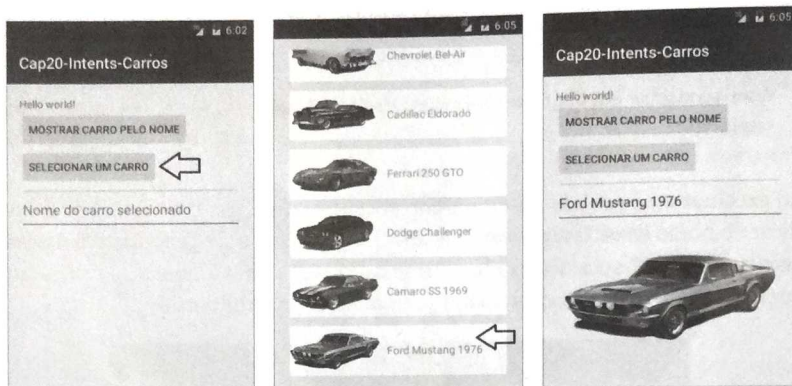


Figura 20.3 – Intent customizada para selecionar um carro.

20.9 Verificando se uma intent será encontrada

Sempre que você disparar uma intent implícita, que geralmente corresponde a uma ação customizada, é recomendado verificar antes de enviar a mensagem se existe alguma aplicação que consegue tratá-la. Isso é preciso porque, caso seja disparada uma intent e nenhuma aplicação consiga tratá-la, vai acontecer um erro em tempo de execução com a seguinte mensagem no LogCat:

No Activity found to handle Intent {descrição da intent aqui}

Faça o teste! Dispare uma intent como essa e veja o erro que vai acontecer.

```
startActivity(new Intent("NAO EXISTE"));
```

Para verificar se a intent pode ser tratada por alguma aplicação, podemos utilizar a seguinte classe:



IntentUtils.java

```
package livroandroid.lib.utils;
public class IntentUtils {
    public static boolean isAvailable(Context ctx, Intent intent) {
        final PackageManager mgr = ctx.getPackageManager();
        List<ResolveInfo> list =
            mgr.queryIntentActivities(intent, PackageManager.MATCH_DEFAULT_ONLY);
        return list.size() > 0;
    }
}
```

Assim, antes de disparar uma intent, o recomendado é sempre verificar se ela pode ser tratada, conforme mostra este código:

```
public void onClickSelecionarCarro(View view) {  
    Uri uri = Uri.parse("carros://br.com.livroandroid.carros/carros");  
    Intent intent = new Intent(Intent.ACTION_PICK, uri);  
    if(IntentUtils.isAvailable(this,intent)) {  
        startActivityForResult(intent, 99);  
    }  
}
```

20.10 Interceptando aplicações nativas

O Android apresenta uma arquitetura orientada a mensagens, e podemos interceptar várias mensagens nativas. Por exemplo, podemos interceptar uma mensagem quando um SMS é entregue, o Wi-Fi é ligado ou desligado, um novo dispositivo Bluetooth é encontrado etc. Ao longo do livro, vamos estudar mais desses exemplos, e você vai acostumando-se com a ideia.

Mas por ora gostaria apenas de explicar mais um recurso interessante do Android. É possível substituir uma aplicação nativa do sistema operacional. Para isso, basta criar uma activity que declare exatamente a mesma ação que ela precisa para ser disparada.

Por exemplo, no aplicativo dos carros tratamos a seguinte mensagem:

```
Uri uri = Uri.parse("carros://br.com.livroandroid.carros/carros");  
Intent intent = new Intent(Intent.ACTION_PICK, uri);
```

Foi definido o atributo `<data android:scheme="carros" />` no arquivo de manifesto, assim essa intent vai entender as mensagens que começarem por `carros://`. Mas como podemos tratar uma mensagem que é enviada para o browser?

```
Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse("http://google.com"));  
startActivity(intent);
```

Se você já entendeu, sabe que basta declarar um filtro com o `<data android:scheme="http" />`, conforme demonstrado a seguir.

```
<intent-filter>  
    <action android:name="android.intent.action.VIEW" />  
    <category android:name="android.intent.category.DEFAULT" />  
    <data android:scheme="http" />  
</intent-filter>
```

Faça o teste! Se você declarar esse filtro em qualquer activity sua, ao disparar a intent para abrir o browser, o Android vai lhe perguntar com qual aplicação de browser você deseja abrir o conteúdo. A figura 20.4 mostra o resultado.

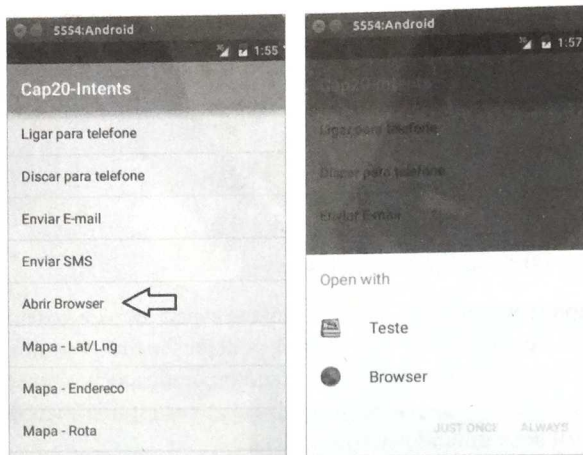


Figura 20.4 – Duas aplicações que podem ser o browser.

Bem, antes que você pergunte, não é possível dar prioridade para nenhuma aplicação. Sendo assim, sempre que existirem duas ou mais aplicações que podem tratar a mesma mensagem, o Android vai perguntar ao usuário para decidir qual aplicação ele deseja abrir. Lembre-se: é o usuário que manda no celular dele, e não você. Para dar outros exemplos, é possível interceptar o botão que chama a activity que faz a ligação.

```
<intent-filter>
    <action android:name="android.intent.action.CALL_BUTTON" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

Ou podemos interceptar a aplicação da câmera, a fim de criar uma aplicação turbinada de câmera.

```
<intent-filter>
    <action android:name="android.media.action.IMAGE_CAPTURE" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

Também podemos interceptar o toque no botão Home, e criar uma tela inicial totalmente customizada.

```
<intent-filter>
  <action android:name="android.intent.action.MAIN" />
  <category android:name="android.intent.category.HOME" />
  <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

Insira esses filtros em alguma activity para testar e veja que interessante. Naturalmente, isso serve apenas para você entender o conceito da plataforma, pois é necessário um motivo muito bom para substituir uma aplicação nativa.

20.11 Lendo código de barras

Para brincar um pouco mais com intents e a integração com outra aplicação, vamos criar um exemplo para ler um código de barras. Como a implementação desse tipo de aplicação não é trivial, podemos simplesmente chamar um aplicativo especialista no assunto a fim de apenas obter o resultado da leitura do código de barras.

Essa aplicação de leitura de código de barras pode ser encontrada no Google Play com o nome Barcode Scanner; depois de instalada, já podemos enviar mensagens para ela. Ela é construída com a biblioteca ZXing (<https://github.com/zxing/zxing>), que contém código aberto e é utilizada por quase 100% de todos os aplicativos que precisam fazer leitura de código de barras. Muitos desenvolvedores pegam o código-fonte dessa biblioteca e o customizam e utilizam dentro do próprio aplicativo.

Para continuar o exemplo a seguir, certifique-se de instalar o aplicativo Barcode Scanner em seu celular. Segue link do Google Play:

<https://play.google.com/store/apps/details?id=com.google.zxing.client.android>

A fim de brincar com a leitura de código de barras, crie um novo projeto que vai conter apenas um simples botão na tela, que vai chamar a aplicação do Barcode Scanner para fazer a leitura. Se preferir, abra o projeto **HelloBarcodeReader** disponível nos exemplos do livro.

A seguir temos um exemplo do método que dispara a intent:

```
public void onClick(View v) {
    Intent intent = new Intent("com.google.zxing.client.android.SCAN");
    startActivityForResult(intent, 0);
}
```

Um jeito melhor ainda de disparar essa intent é verificar se o aplicativo está instalado. Caso não esteja, vamos abrir o Google Play para o usuário instalar. Veja que tudo é feito por intents.


```

public void onClick(View v) {
    Intent intent = new Intent("com.google.zxing.client.android.SCAN");
    if(IntentUtils.isAvailable(context,intent)) {
        startActivityForResult(intent, 0);
    } else {
        Toast.makeText(context, "Instale o app Barcode Scanner", Toast.LENGTH_SHORT).show();
        intent = new Intent(Intent.ACTION_VIEW);
        intent.setData(Uri.parse("market://details?id=com.google.zxing.client.android"));
        startActivity(intent);
    }
}

```

Para essa intent funcionar, declare a permissão do ZXing no manifesto:

```
<uses-permission android:name="com.google.zxing.client.android.SCAN" />
```

Dessa forma a aplicação de leitura de código de barras será executada e, ao apontar a câmera para um código de barras válido, ele será lido e identificado. Recomendo você testar com QR Code, pois códigos de barra 2D (exemplo: boleto) requerem algumas customizações para serem lidos. Para gerar um código de barras a fim de testar este exemplo, você pode utilizar este link <http://zxing.appspot.com/generator/>.

Feito isso, o número será retornado para a aplicação de origem por meio do método `onActivityResult(codigo, resultado, intent)`.

```

public void onActivityResult(int requestCode, int resultCode, Intent intent) {
    if (requestCode == 0) {
        if (resultCode == RESULT_OK) {
            String numero = intent.getStringExtra("SCAN_RESULT");
            // Este é o número do código de barras
        }
    }
}

```

Deixo como exercício para você construir esse aplicativo, ou caso prefira abra o projeto de exemplo deste capítulo. O nome do projeto é **HelloBarcodeReader**.

Nota: o recurso de enviar mensagens para todas as aplicações instaladas e ainda ler o resultado torna o desenvolvimento de aplicações para o Android muito flexível, facilitando muito a integração entre diferentes aplicações, uma vez que uma pode chamar a outra com facilidade.

20.12 Nomenclatura das intents

Repare que uma das intents que utilizamos anteriormente tinha a seguinte ação:

```
br.com.livroandroid.intents.TESTE
```

E no exercício sobre código de barras vimos que a intent do ZXing é:

```
com.google.zxing.client.android.SCAN
```

Este breve tópico serve apenas para alertá-lo da importância do padrão de nomes de uma intent. Por convenção, o nome do pacote da aplicação sempre é colocado antes da ação, com o objetivo de diferenciar as intents de diferentes aplicações. No aplicativo dos carros mostramos como abrir uma activity com base na seguinte intent:

```
// Abre o carro pelo nome
Uri uri = Uri.parse("carros://br.com.livroandroid.carros/carros/Ferrari FF");
Intent intent = new Intent(Intent.ACTION_VIEW, uri);
startActivity(intent);
```

Neste caso estamos utilizando a constante `Intent.ACTION_VIEW` de uma ação nativa do Android, mas essa ação também segue a mesma convenção, pois sua string é `android.intent.action.VIEW`.

20.13 Links úteis

Para continuar seus estudos sobre a classe `Intent`, separei alguns links da documentação oficial.

- **API Guides – Intents and Intent Filters**

<http://developer.android.com/guide/components/intents-filters.html>

- **API Guides – Common Intents**

<http://developer.android.com/guide/components/intents-common.html>

- **Android Training – Sending the User to Another App**

<http://developer.android.com/training/basics/intents/sending.html>

- **Android Training – Allowing Other Apps to Start Your Activity**

<http://developer.android.com/training/basics/intents/filters.html>



CAPÍTULO 21

Multimídia – áudio, vídeo e câmera

Neste capítulo, vamos estudar os recursos de multimídia disponíveis no Android, como mostrar vídeos, reproduzir áudio e utilizar a câmera para tirar fotos e gravar vídeos.

21.1 Formatos de áudio e vídeo suportados

Segundo a documentação oficial do Android, os seguintes formatos são suportados ao desenvolver as aplicações.

- **Áudio** – mp3, midi, 3gp, ogg, m4a, wav e flac.
- **Vídeo** – mp4 e 3gp.

Os protocolos de comunicação suportados são HTTP, HTTPS e RTSP. O Real Time Streaming Protocol (RTSP) é extremamente recomendado no caso de streaming de vídeos. Para mais informações consulte o site oficial:

<http://developer.android.com/guide/appendix/media-formats.html>

21.2 Classe Media Player

Vamos começar este capítulo brincando um pouco e criando um player de mp3. Para isso, vamos utilizar a classe `android.media.MediaPlayer`, que é responsável por reproduzir qualquer recurso de mídia, como áudio e vídeo.

Tocar uma música é simples, basta configurar a origem do arquivo com alguma versão do método `setDataSource(...)`, chamar o método `prepare()` a fim de preparar a mídia e na sequência o método `start()`. O método `prepare()` é síncrono e só retorna quando o arquivo estiver pronto para reprodução. Depois que a mídia

estiver pronta para ser reproduzida, basta chamar o método `start()` para iniciar a música ou vídeo.

O exemplo a seguir mostra como tocar uma música a partir de um caminho de arquivo conhecido.

```
MediaPlayer player = new MediaPlayer();
player.setDataSource(Uri.parse("/storage/sdcard/Music/linkin_park1.mp3"));
player.prepare();
player.start();
```

Veja que sempre é necessário chamar a sequência de métodos `setDataSource(...) > prepare() > start()`. Para simplificar esse fluxo, existe o método `MediaPlayer.create(...)`, que é um atalho que automaticamente configura a `dataSource` e chama o `prepare()`. Portanto, basta você chamar o método `start()`.

```
MediaPlayer player = MediaPlayer.create(this, Uri.parse("http://site.com.br/musica.mp3"));
player.start();
```

O método `MediaPlayer.create(...)` é útil no caso de você precisar tocar uma música que está na `/res/raw` do projeto, pois internamente esse método encapsula o código necessário para configurar a `dataSource`. O exemplo de código a seguir mostra como tocar a música `/res/raw/musica.mp3`.

```
MediaPlayer player = MediaPlayer.create(this, R.raw.musica);
player.start();
```

Depois que a música estiver tocando, é possível chamar o método `stop()` para parar a reprodução ou o `pause()` para fazer apenas uma breve pausa. Se o método `pause()` for chamado, a música é interrompida, mas pode ser reiniciada normalmente mais tarde. Para reiniciá-la, basta chamar o método `start()` novamente. O método `stop()` deve ser chamado para parar completamente a reprodução da música. Se depois da chamada do método `stop()` você quiser voltar a reproduzir a música, é necessário reiniciar o `MediaPlayer`. Para isso, o método `reset()` deve ser chamado, fazendo com que o `MediaPlayer` volte a seu estado original. Consequentemente, os métodos `setDataSource(arquivo)`, `prepare()` e `start()` precisam ser chamados novamente para reproduzir o arquivo.

O último método que deve ser explicado é o `release()`, o qual deve ser chamado para liberar a memória e os recursos do `MediaPlayer` quando não for mais necessário usar essa classe. Se você tem certeza de que depois do `stop()` não voltará a reproduzir a música, pode chamar o `release()`.

Para assegurar-se de que o método `release()` será chamado, é possível chamá-lo no método `onDestroy()` da `activity` para garantir que a reprodução da música ou vídeo tenha sido encerrada e os recursos liberados, conforme demonstrado a seguir:

```

@Override
protected void onDestroy() {
    super.onDestroy();
    if (player != null) {
        player.stop();
        player.release();
    }
}

```

A figura 21.1 mostra o diagrama de estado da classe `MediaPlayer`. Observe atentamente a ordem em que os métodos devem ser chamados:

Fonte: <http://developer.android.com/reference/android/media/MediaPlayer.html>

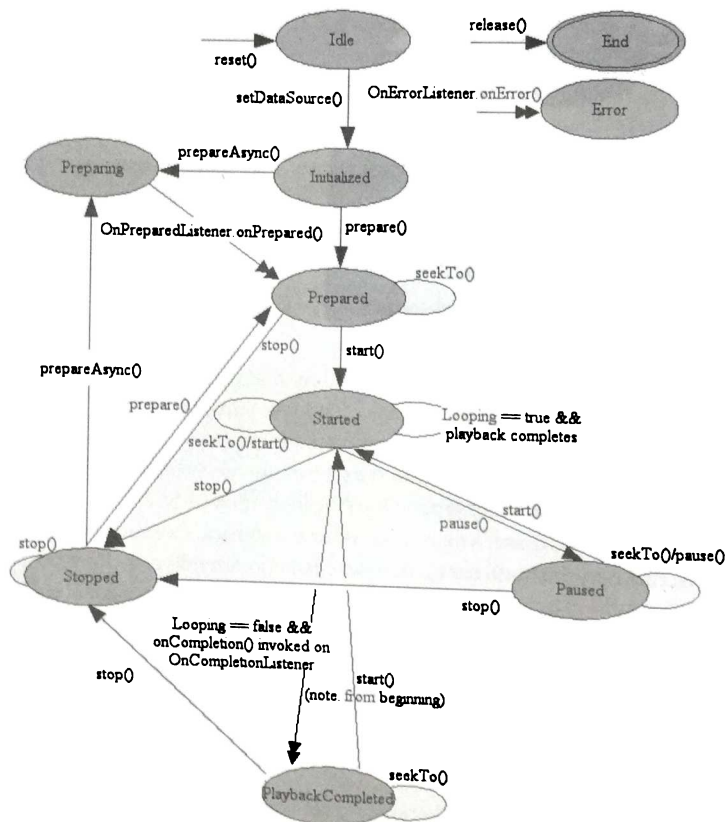


Figura 21.1 – Diagrama com os métodos do `MediaPlayer`.

21.3 Criando um player de mp3

Para o próximo exemplo, criaremos o projeto **PlayerMp3**.

Antes de escrever o código da activity que exibirá a tela para o usuário, criaremos uma classe chamada **PlayerMp3**, que encapsulará o acesso à classe **MediaPlayer** para controlar a reprodução das músicas e facilitar o exemplo.



PlayerMp3

. . .

```
public class PlayerMp3 implements OnCompletionListener {
    private static final String CATEGORIA = "livroandroid";
    private static final int NOVO    = 0;
    private static final int TOCANDO = 1;
    private static final int PAUSADO = 2;
    private static final int PARADO  = 3;
    // Começa o status zerado
    private int status = NOVO;
    private MediaPlayer player;
    // Caminho da música
    private String mp3;
    public PlayerMp3() {
        // Cria o MediaPlayer
        player = new MediaPlayer();
        // Executa o listener quando terminar a música
        player.setOnCompletionListener(this);
    }
    public void start(String mp3) {
        this.mp3 = mp3;
        try {
            switch (status) {
                case TOCANDO:
                    player.stop();
                case PARADO:
                    player.reset();
                case NOVO:
                    player.setDataSource(mp3);
                    player.prepare();
                case PAUSADO:
                    player.start();
                    break;
            }
            status = TOCANDO;
        }
    }
}
```

```

    } catch (Exception e) {
        Log.e(CATEGORIA, e.getMessage(), e);
    }
}

public void pause() {
    player.pause();
    status = PAUSADO;
}

public void stop() {
    player.stop();
    status = PARADO;
}

// Encerra o MediaPlayer e libera a memória
public void close() {
    stop();
    player.release();
    player = null;
}

/**
 * @see android.media.MediaPlayer.OnCompletionListener#onCompletion(android.media.MediaPlayer)
 */
public void onCompletion(MediaPlayer mp) {
    Log.d(CATEGORIA, "Fim da música: " + mp3);
}

public boolean isPlaying() {
    // Retorna true se a música está tocando ou se está em PAUSE
    return status == TOCANDO || status == PAUSADO;
}
}

```

Observe que essa classe controla o estado de reprodução da mídia. Para isso, o atributo `status` da classe é utilizado, de modo que a sequência correta dos métodos seja chamada, respeitando o diagrama de estados da classe `MediaPlayer`. Para tal, o método `start(mp3)` verifica o estado atual antes de iniciar a reprodução.

O próximo passo é criar o arquivo de layout, que contém um campo de texto com o caminho do arquivo `mp3` e os botões **start**, **pause** e **stop** para controlar o áudio.



/res/layout/activity_main.xml

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" android:padding="16dp">

```

```

<TextView
    android:layout_width="match_parent" android:layout_height="wrap_content"
    android:text="Player MP3" />
<EditText
    android:id="@+id/arquivo"
    android:layout_width="match_parent" android:layout_height="wrap_content"
    android:text="/storage/sdcard/Music/linkin_park1.mp3" />
<LinearLayout
    android:layout_width="match_parent" android:layout_height="wrap_content" >
    <ImageButton
        android:id="@+id/start"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:src="@drawable/img_start" />
    <ImageButton
        android:id="@+id/pause"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:src="@drawable/img_pause" />
    <ImageButton
        android:id="@+id/stop"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:src="@drawable/img_stop" />
</LinearLayout>
</LinearLayout>

```

A seguir, podemos visualizar a activity que usará esse arquivo de layout e tratará os eventos dos botões para tocar a música. Todo o trabalho, na verdade, é delegado para a classe `PlayerMp3` que criamos anteriormente.



MainActivity.java

```

...
public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    private static final String TAG = "livro";
    private PlayerMp3 player = new PlayerMp3();
    private EditText text;
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_main);
        text = (EditText) findViewById(R.id.arquivo);
        findViewById(R.id.start).setOnClickListener(this);
        findViewById(R.id.pause).setOnClickListener(this);
        findViewById(R.id.stop).setOnClickListener(this);
    }
}

```



```

public void onClick(View view) {
    try {
        if (view.getId() == R.id.start) {
            String mp3 = text.getText().toString();
            player.start(mp3);
        } else if (view.getId() == R.id.pause) {
            player.pause();
        } else if (view.getId() == R.id.stop) {
            player.stop();
        }
    } catch (Exception e) {
        Log.e(TAG, e.getMessage(), e);
        Toast.makeText(this, e.getMessage(), Toast.LENGTH_SHORT).show();
    }
}

@Override
protected void onDestroy() {
    super.onDestroy();
    // Libera recursos do MediaPlayer
    player.close();
}
}

```

Neste exemplo, o método `onClick(view)` é chamado para tratar o evento de quando o usuário clica em algum botão da tela. Nesse método, o caminho do arquivo mp3 digitado no formulário da tela é recuperado para tocar a música. Ao executar a aplicação, a tela será visualizada conforme a figura 21.2, e você pode usar os botões de **play**, **pause** e **stop** para tocar o mp3. A figura ao lado do player mp3 mostra um print da janela **File Explorer** do Android Studio, pois este exemplo lê o arquivo do SD card. Certifique-se de colocar os arquivos de mídia no SD card do emulador conforme indicado.

Nota: neste exemplo, se você tocar a música e fechar a tela da activity, o `PlayerMp3` será encerrado, pois ele está atrelado ao ciclo de vida da activity. No capítulo 27, sobre a classe `Service`, vamos aprender a executar processos em segundo plano que podem ficar vivos no sistema operacional do Android. Utilizando esses serviços de segundo plano, a música continuará executando mesmo ao fechar a aplicação.

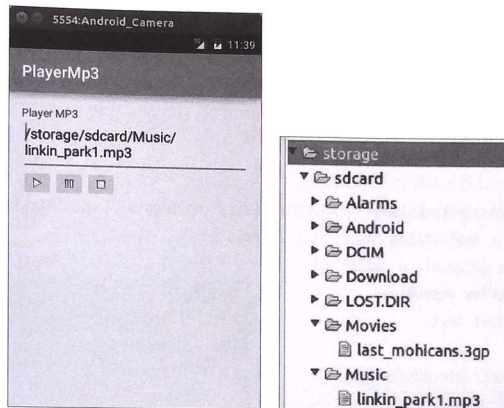


Figura 21.2 – Exemplo completo para tocar a música.

21.4 Reproduzindo vídeo com a classe `VideoView`

A classe `MediaPlayer` consegue reproduzir qualquer arquivo de mídia, seja áudio ou vídeo. O vídeo, porém, precisa ser renderizado em alguma view da tela, e para simplificar esse trabalho foi criada a classe `VideoView`, que é uma subclasse de `View`, portanto, basta inseri-la no layout para mostrar o vídeo. A classe `VideoView` encapsula o acesso ao `MediaPlayer` e de resto ela é igual.

Para configurar o conteúdo do vídeo, ou seja, qual arquivo ou URL que será mostrado, você pode utilizar os métodos `setVideoPath(string)` ou o método `setVideoURI(uri)`. Para demonstrar a utilização do `VideoView`, vamos criar um exemplo idêntico ao **PlayerMp3**, mas desta vez vamos chamar de **PlayerVideo**. Para continuar o exercício, crie um projeto chamado **PlayerVideo** com o seguinte arquivo de layout:

```
res/layout/activity_main.xml

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout android:orientation="vertical" . . .>
    // EditText para digitar o caminho do vídeo aqui
    // Botões de play, pause e stop aqui
    // Por último o VideoView
    <VideoView android:id="@+id/videoView"
        android:layout_width="match_parent"
        android:layout_height="0dp" android:layout_weight="1" />
</LinearLayout>
```

Estou simplificando o layout, pois ele é o mesmo do exemplo anterior. A única diferença é que o `VideoView` foi inserido abaixo dos botões de **play**, **pause**, **stop**. No código da `activity`, vamos recuperar o `VideoView` pelo `id` e chamar os métodos para mostrar o vídeo.



MainActivity.java

```

...
public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    private static final String TAG = "livroandroid";
    // Classe que encapsula o MediaPlayer
    private VideoView videoView;
    private EditText text;
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_main);
        text = (EditText) findViewById(R.id.arquivo);
        videoView = (VideoView) findViewById(R.id.videoView);
        videoView.setMediaController(new MediaController(this));
        findViewById(R.id.start).setOnClickListener(this);
        findViewById(R.id.pause).setOnClickListener(this);
        findViewById(R.id.stop).setOnClickListener(this);
    }
    public void onClick(View view) {
        try {
            if (view.getId() == R.id.start) {
                String path = text.getText().toString();
                videoView.setVideoURI(Uri.parse(path));
                videoView.start();
            } else if (view.getId() == R.id.pause) {
                videoView.pause();
            } else if (view.getId() == R.id.stop) {
                videoView.stopPlayback();
            }
        } catch (Exception e) {
            Log.e(TAG, e.getMessage(), e);
            Toast.makeText(this, e.getMessage(), Toast.LENGTH_SHORT).show();
        }
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
    }
}

```

```
// Libera recursos do MediaPlayer
videoView.stopPlayback();
}
}
```

A figura 21.3 mostra o resultado com o vídeo executando, mas lembre-se de colocá-lo no SD card do emulador. O interessante do código é o método `setMediaController(controller)`, o qual adiciona os controles da reprodução do vídeo, que podem ser vistos sobre o `VideoView` na parte inferior da tela.

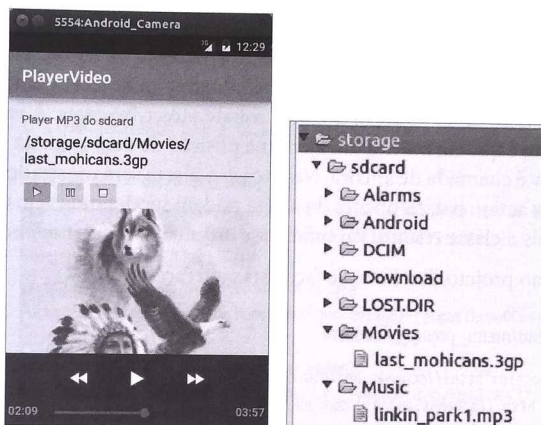


Figura 21.3 – VideoView.

21.5 Utilizando uma intent e o player de vídeo nativo

Já sabemos que para mostrar um vídeo podemos utilizar a classe `VideoView`, mas para isso é necessário criar um layout, criar uma nova activity e fragment etc. Mas por que não mostramos o vídeo disparando uma simples intent?

Mais uma vez, as intents vão dar o ar de sua graça, pois a maneira mais simples de mostrar um vídeo no Android é disparar uma intent para o player de vídeo nativo. Por exemplo, se você conhecer o caminho do arquivo, URL ou URI do vídeo, basta disparar a seguinte intent, que será tratada pelo player de vídeo nativo.

```
Uri uri = Uri.parse(url);
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.setDataAndType(uri, "video/*");
context.startActivity(intent);
```

Pronto! Tem algo melhor do que mostrar o vídeo no player nativo do Android? A vantagem é que o usuário inclusive já está acostumado com o player. Quando o vídeo terminar, sua aplicação volta a executar e pronto. Simples assim.

No projeto dos carros, vamos brincar um pouco e por motivos de aprendizado ofereceremos ao usuário três opções para tocar o vídeo:

1. Mostrar o vídeo no browser.
2. Mostrar o vídeo no player nativo do Android.
3. Abrir uma nova activity com o `VideoView`.

O fragment que mostra os detalhes do carro apresenta os botões **Vídeo** e **Mapa** na action bar. Ao tocar no botão **Vídeo**, vamos mostrar uma alerta com essas três opções. Para isso, vamos utilizar a classe `android.widget.PopupMenu`. A vantagem da classe `PopupMenu` é que ela mostra um alerta na posição de determinada view, sendo que essa view é chamada de âncora. Neste caso o alerta será mostrado ao lado do botão **Vídeo** da action bar. As opções do alerta podem ser definidas em um arquivo de menu, pois a classe `PopuMenu` vai inflar esse arquivo para criar a lista.

Agora volte ao projeto dos carros e faça estas alterações:

 `/res/menu/menu_popup_video.xml`

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item android:id="@+id/action_video_browser"
        android:title="@string/action_video_browser" app:showAsAction="always" />
    <item android:id="@+id/action_video_player" android:title="@string/action_video_player"
        app:showAsAction="always" />
    <item android:id="@+id/action_video_videoview"
        android:title="@string/action_video_videoview" app:showAsAction="always" />
</menu>
```

 `/res/values/strings.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    ...
    <string name="action_video_browser">Browser</string>
    <string name="action_video_player">Video Player</string>
    <string name="action_video_videoview">VideoView</string>
</resources>
```

Agora que criamos o arquivo de menu, vamos alterar o código que trata o evento da action bar para, ao selecionar o item **Vídeo**, mostrar as opções para o usuário com a ajuda da classe `PopupMenu`. Observe que o código utiliza a classe `android.support.v7.widget.PopupMenu` da biblioteca de compatibilidade.



CarroFragment.java

```
...
import android.support.v7.widget.PopupMenu;
import livroandroid.lib.utils.IntentUtils;
public class CarroFragment extends BaseFragment {
    ...
    public boolean onOptionsItemSelected(MenuItem item) {
        ...
        } else if (item.getItemId() == R.id.action_video) {
            // URL do vídeo
            final String url = carro.urlVideo;
            // Lê a view que é a âncora do popup (é a view do botão da action bar)
            View menuItemView = getActivity().findViewById(item.getItemId());
            if (menuItemView != null && url != null) {
                // Cria o PopupMenu posicionado na âncora
                PopupMenu popupMenu = new PopupMenu(getActivity().getThemedContext(),
                    menuItemView);
                popupMenu.inflate(R.menu.menu_popup_video);
                popupMenu.setOnMenuItemClickListener(new PopupMenu.OnMenuItemClickListener() {
                    @Override
                    public boolean onMenuItemClick(MenuItem item) {
                        if (item.getItemId() == R.id.action_video_browser) {
                            // Abre o vídeo no browser
                            IntentUtils.openBrowser(getActivity(), url);
                        } else if (item.getItemId() == R.id.action_video_player) {
                            // Abre o vídeo no Player de Vídeo Nativo
                            IntentUtils.showVideo(getActivity(), url);
                        } else if (item.getItemId() == R.id.action_video_videoview) {
                            // Abre outra activity com VideoView
                        }
                        return true;
                    }
                });
                popupMenu.show();
            }
        }
        return super.onOptionsItemSelected(item);
    }
}
```

Para o código compilar, importe a classe `IntentUtils` da biblioteca `android-utils`, pois ela contém os métodos para disparar as intents, a fim de mostrar o vídeo no browser ou no player nativo. Para sua consulta, o código-fonte dessa classe pode ser visualizado a seguir:



`IntentUtils.java`

```
package livroandroid.lib.utils;

public class IntentUtils {
    private static final String TAG = "IntentUtils";
    . . .
    public static void openBrowser(Context context, String url) {
        try {
            Uri uri = Uri.parse(url);
            Intent intent = new Intent(Intent.ACTION_VIEW, uri);
            context.startActivity(intent);
        } catch (ActivityNotFoundException e) {
            Log.e(TAG, "openBrowser() - ActivityNotFoundException [" + url + "]: " + e.getMessage());
        }
    }
    public static void showVideo(Context context, String url) {
        try {
            Uri uri = Uri.parse(url);
            Intent intent = new Intent(Intent.ACTION_VIEW);
            intent.setDataAndType(uri, "video/*");
            context.startActivity(intent);
        } catch (ActivityNotFoundException e) {
            Log.e(TAG, "showVideo() - ActivityNotFoundException [" + url + "]: " + e.getMessage());
        }
    }
}
```

Com essas alterações, ao executar o projeto e selecionar a ação **Vídeo** na action bar, o alerta com as opções será mostrado conforme a figura 21.4.

Ao selecionar a opção **Browser**, uma intent é disparada para abrir o browser. Assim, o vídeo será aberto da mesma forma como se você tivesse aberto o browser e digitado uma URL de um arquivo vídeo no endereço. Na prática, você só precisa utilizar essa opção se não souber ao certo qual será o tipo do conteúdo, então você simplesmente dispara a intent.

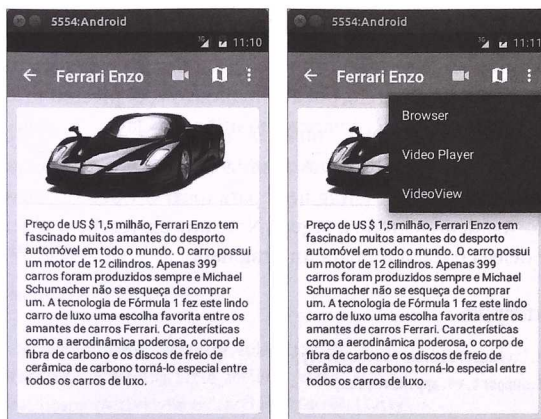


Figura 21.4 – Alerta com opções do vídeo.

Mas caso saiba que se trata de uma URL de vídeo, a opção do player de vídeo nativo é mais interessante, pois dispara uma intent que será tratada pela aplicação nativa de vídeo. Vale relembrar que, se mais de uma aplicação conseguir tratar a mensagem, o Android vai perguntar ao usuário qual aplicação deve executar. A figura 21.5 mostra um vídeo sendo tocado no player nativo do emulador.



Figura 21.5 – Player de vídeo nativo.

21.6 Utilizando o `VideoView` no projeto dos carros

A forma mais simples de mostrar um vídeo na aplicação é disparar uma intent e utilizar o player nativo. Se, porém, por algum motivo você precisar mostrar o vídeo dentro do layout da aplicação, é necessário utilizar a classe `VideoView`.

Como já estudamos a classe `VideoView`, vamos só exercitar o conceito mais uma vez e criar uma nova activity e fragment para tocar o vídeo do carro. Crie uma activity conforme demonstrado a seguir. Note que ela recebe o objeto carro por parâmetro e delega o trabalho para um fragment.



`VideoActivity.java`

```
package br.com.livroandroid.carros.activity;
import android.support.v4.app.NavUtils;
...
import br.com.livroandroid.carros.fragments.VideoFragment;
public class VideoActivity extends BaseActivity {
    private Carro carro;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_video);
        // Configura a Toolbar como a action bar
        setUpToolbar();
        carro = (Carro) getIntent().getSerializableExtra("carro");
        getSupportActionBar().setTitle(carro.nome);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        if (savedInstanceState == null) {
            // Adiciona o fragment ao layout da activity
            VideoFragment videoFragment = new VideoFragment();
            videoFragment.setArguments(getIntent().getExtras());
            getSupportFragmentManager().beginTransaction().replace(R.id.fragLayout,
                videoFragment).commit();
        }
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case android.R.id.home:
                Intent intent = NavUtils.getParentActivityIntent(getActivity());
                intent.putExtra("carro", carro);
                NavUtils.navigateUpTo(getActivity(), intent);
                return true;
        }
    }
}
```

```

    }
    return super.onOptionsItemSelected(item);
}
}

```

Lembre-se de que todas as activities precisam ser configuradas no manifesto. Repare que, no caso desta activity do vídeo, estou configurando a activity pai como CarroActivity, e não MainActivity.



AndroidManifest.xml

```

<application . . .>
    . . .
    <activity android:name=".activity.VideoActivity"
        android:label="@string/title_activity_video"
        android:parentActivityName=".activity.CarroActivity" >
        <meta-data android:name="android.support.PARENT_ACTIVITY"
            android:value=".activity.CarroActivity" />
    </activity>
</application>

```

No código da activity, perceba que estou interceptando a ação do botão voltar do up navigation (android.R.id.home) e utilizando a classe NavUtils para voltar para a activity anterior, que está acima na hierarquia. Lembre-se de que a activity pai da VideoActivity é a CarroActivity e não a MainActivity. Por padrão, o funcionamento do up navigation é não passar nenhum parâmetro para a activity pai, e como a classe CarroActivity recebe o carro por parâmetro, precisamos customizar esse retorno.

Entenda que na prática todo esse código poderia ser substituído por um simples finish(), pois dessa forma, quando a activity do vídeo terminar, a activity do carro passaria a ocupar o topo da pilha. Propositalmente, mostrei esse código que utiliza a classe NavUtils para demonstrar casos mais complexos. Digamos que você tivesse a seguinte pilha de activities:

```
> Main Activity > A > B > C > D > E.
```

Se você estiver na activity E, como fazemos para voltar para a activity B? Neste caso, se utilizarmos o método finish(), apenas a activity E será finalizada e a activity D passará a ocupar o topo da pilha. Mas o que queremos é derrubar todas as activities da pilha, até chegar na B. Utilizamos então o flag FLAG_ACTIVITY_CLEAR_TOP na intent, mas a classe NavUtils contém métodos justamente para facilitar esse tipo de navegação.

Vamos continuar com o código para mostrar o vídeo do carro. A seguir, podemos ver o arquivo de layout da activity VideoActivity, que contém apenas a Toolbar e um FrameLayout para adicionar o fragment.



/res/layout/activity_video.xml

```
<LinearLayout android:orientation="vertical"
    android:layout_width="match_parent" android:layout_height="match_parent" >
    <include layout="@layout/include_toolbar" />
    <FrameLayout
        android:id="@+id/fragLayout"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        tools:layout="@layout/fragment_video" />
</LinearLayout>
```

Nota: lembre-se de que todas as activities precisam incluir a Toolbar no layout.

Mais uma vez, estamos utilizando a activity para controlar a navegação das telas, mas internamente todo o conteúdo e lógica ficam no fragment. A classe `VideoFragment` precisa ler o objeto carro que é enviado pelos argumentos e configurar o `VideoView` com a URL do vídeo.



VideoFragment.java

```
package br.com.livroandroid.carros.fragments;
import android.widget.MediaController;
...
public class VideoFragment extends BaseFragment {
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_video, container, false);
        VideoView videoView = (VideoView) view.findViewById(R.id.videoView);
        Carro c = (Carro) getArguments().getSerializable("carro");
        if (c != null) {
            videoView.setVideoURI(Uri.parse(c.urlVideo));
            videoView.setMediaController(new MediaController(getContext()));
            videoView.start();
            toast("start: " + c.urlVideo);
        }
        return view;
    }
}
```

O arquivo de layout do fragment contém apenas um `VideoView`.



/res/layout/fragment_video.xml

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent" >
    <VideoView android:id="@+id/videoView"
        android:layout_width="match_parent" android:layout_height="match_parent" />
</FrameLayout>
```

Agora que já criamos a activity e o fragment que vai mostrar o vídeo, volte à classe CarroFragment e altere o trecho de código do PopupMenu cujo alerta inflamos com as opções. Na opção do VideoView, vamos navegar para a activity que acabamos de criar.



CarroFragment.java

```
public class CarroFragment extends BaseFragment {
    . . .
    popupMenu.setOnMenuItemClickListener(new PopupMenu.OnMenuItemClickListener() {
        @Override
        public boolean onMenuItemClick(MenuItem item) {
            if (item.getItemId() == R.id.action_video_browser) { . . . }
            else if (item.getItemId() == R.id.action_video_player) { ... }
            else if (item.getItemId() == R.id.action_video_videoview) {
                // Abre outra activity com VideoView
                Intent intent = new Intent(getContext(), VideoActivity.class);
                intent.putExtra("carro", carro);
                startActivity(intent);
            }
            return true;
        }
    });
}
```

Feito isso, está pronto. Desta vez, ao selecionar a opção **Vídeo** e depois o item **VideoView**, vamos navegar para a activity do vídeo que acabamos de criar. A figura 21.6 mostra o resultado com o vídeo sendo mostrado no VideoView.

Com relação ao projeto dos carros, isso é tudo, pois já mostrei três maneiras para mostrar o vídeo do carro. Particularmente, sempre tento convencer meus clientes de que a melhor opção é disparar uma intent e utilizar o próprio player nativo. O motivo é simples, pois é mais fácil de fazer e o resultado é muito melhor. Mas é claro que tudo depende do caso. Por exemplo, em uma aplicação para tablet em que temos um grande espaço disponível na tela, muitas vezes opta-se por utilizar um VideoView, a fim de aproveitar melhor o espaço das views dentro do layout.

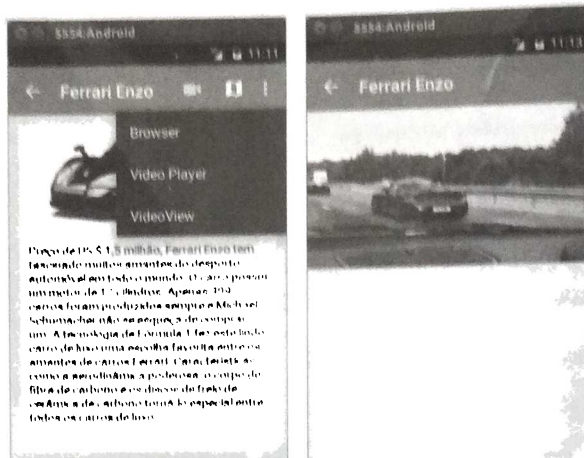


Figura 21.6 – Vídeo do carro no VideoView.

21.7 Tirando fotos com uma intent

Se você acha que nossa conversa sobre as intents tinha chegado ao fim, está enganado, pois, como disse no capítulo anterior, as intents são o coração do Android e são utilizadas em todos os lugares.

Para tirar uma foto com o Android, existem duas opções.

A primeira é utilizar a classe `android.hardware.camera` e controlar a API de baixo nível. Neste caso, é necessário implementar alguns métodos para mostrar a superfície da câmera, além de controlar seu estado, rotação de tela etc. Enfim, é algo que exige um pouco de código. Mas no Android muitas vezes não precisamos desenvolver tudo do zero, podemos apenas integrar aplicações que já existem. Portanto, a segunda opção consiste em disparar uma intent para a aplicação nativa da câmera, que vai se encarregar de todo o trabalho pesado, e, diga-se de passagem, não existe nenhuma aplicação melhor para tirar fotos que a própria nativa do Android. Como resultado, a aplicação da câmera vai nos devolver um objeto do tipo `Bitmap`, o qual representa uma imagem no Android.

Na prática, disparar uma intent é o recomendado, uma vez que o usuário vai tirar a foto com o aplicativo de câmera com o qual ele já está acostumado. Logo, vamos utilizar essa solução neste livro.

Para continuar, crie um novo projeto chamado **HelloCamera** ou abra o projeto disponível nos exemplos do livro.

A seguir, podemos ver o arquivo de layout da activity. Teremos um botão que vai disparar a câmera com uma intent e um `ImageView` que vai receber o resultado da foto.



/res/layout/activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent" android:layout_height="fill_parent"
    android:orientation="vertical" android:padding="16dp">
    <ImageButton android:id="@+id/btAbrirCamera"
        android:layout_width="60dp" android:layout_height="60dp"
        android:src="@android:drawable/ic_menu_camera" android:text="Abrir a Camera" />
    <ImageView android:id="@+id/imagen" android:layout_width="match_parent"
        android:layout_height="0dp" android:layout_weight="1"
        android:layout_gravity="center" />
</LinearLayout>
```

Veja que o botão que vai disparar a intent está configurado com a imagem `android:src="@android:drawable/ic_menu_camera"`, nativa do Android. A seguir, temos o código da activity que dispara a intent para abrir a câmera:



MainActivity.java

```
...
public class MainActivity extends AppCompatActivity {
    private ImageView imgView;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        imgView = (ImageView) findViewById(R.id.imagen);
        ImageButton b = (ImageButton) findViewById(R.id.btAbrirCamera);
        b.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent i = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
                startActivityForResult(i, 0);
            }
        });
    }
}
```

```

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (data != null) {
        Bundle bundle = data.getExtras();
        if (bundle != null) {
            // Recupera o Bitmap retornado pela câmera
            Bitmap bitmap = (Bitmap) bundle.get("data");
            // Atualiza a imagem na tela
            imageView.setImageBitmap(bitmap);
        }
    }
}
}
}

```

Uma última configuração necessária é declarar a permissão para utilizar a câmera no arquivo de manifesto.



AndroidManifest.xml

```

<manifest . . .>
    <uses-permission android:name="android.permission.CAMERA" />
    <application . . . />
</manifest>

```

Como já estudamos o que é uma intent, acredito que este código seja tranquilo para você. Ao disparar a intent, a câmera vai abrir, e depois de tirar a foto o resultado será entregue no método `onActivityResult(...)`. Como eu disse antes, cada aplicação retorna às informações de uma forma diferente. No caso da aplicação da câmera, o retorno é um objeto do tipo `Bitmap`. Para obter o `Bitmap`, devemos ler o parâmetro "data" do `Bundle`. Esse `Bitmap` é mostrado no `ImageView` que está no layout.

A figura 21.7 mostra o resultado da foto. A primeira parte é o layout antes de tirar a foto, a segunda parte é a aplicação da câmera executando no emulador e a terceira parte é a foto retornada pela câmera do emulador, que retorna sempre a mesma figurinha apenas para simular.

Importante: para tirar fotos com o emulador, é preciso configurá-lo no momento de criar o AVD. Nas configurações da câmera do emulador, selecione o item `Back` (câmera traseira) e escolha a opção `Emulated` para simular uma câmera. Em meus testes, a câmera do emulador com o webcam do computador não funciona bem. Outra opção é tirar fotos com um dispositivo real.

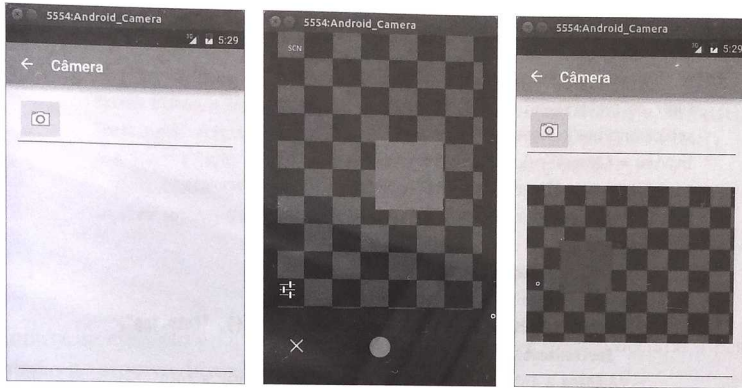


Figura 21.7 – Vídeo do carro no VideoView.

21.8 Tirando fotos – como obter o arquivo da foto

No exemplo anterior, vimos como é simples tirar uma foto utilizando uma intent e delegar o trabalho ao aplicativo nativo da câmera. No emulador, é retornada sempre a mesma figura, mas ao tirar a foto em um dispositivo real você verá que o retorno da câmera é um pequeno thumb da foto, ou seja, uma imagem reduzida da original. O Android faz isso para economizar memória.

Para obter a foto original com toda a resolução, é necessário passar um parâmetro para a intent da câmera, que deve ser o caminho para salvar o arquivo. Neste caso a aplicação da câmera não vai retornar o thumb, pois ela vai simplesmente salvar a foto no caminho de arquivo que foi informado pela intent. E no método `onActivityResult(...)` da activity só é preciso testar se o retorno da câmera foi `RESULT_OK` e ler o arquivo da foto no caminho que nós mesmos informamos.

O código-fonte a seguir utiliza o mesmo arquivo de layout do exemplo anterior, porém mostra como obter a foto com toda a resolução da câmera.



MainActivity.java

```
...
import livroandroid.lib.utils.ImageResizeUtils;
import livroandroid.lib.utils.SDCardUtils;
public class MainActivity extends AppCompatActivity {
    // Caminho para salvar o arquivo
    private File file;
```



```

private ImageView imgView;
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    imgView = (ImageView) findViewById(R.id.imagem);
    ImageButton b = (ImageButton) findViewById(R.id.btAbrirCamera);
    b.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            // (*1*) Cria o caminho do arquivo no SD card
            file = SDCardUtils.getPrivateFile(getBaseContext(), "foto.jpg",
                Environment.DIRECTORY_PICTURES);
            // Chama a intent informando o arquivo para salvar a foto
            Intent i = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
            i.putExtra(MediaStore.EXTRA_OUTPUT, Uri.fromFile(file));
            startActivityForResult(i, 0);
        }
    });
    if (savedInstanceState != null) {
        // (*2) Se girou a tela, recupera o estado
        file = (File) savedInstanceState.getSerializable("file");
        showImage(file);
    }
}
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    // (*3*) Salvar o estado caso gire a tela
    outState.putSerializable("file", file);
}
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (resultCode == RESULT_OK && file != null) {
        // (*4*) Se a câmera retornou, vamos mostrar o arquivo da foto
        showImage(file);
    }
}
// Atualiza a imagem na tela
private void showImage(File file) {
    if(file != null && file.exists()) {
        Log.d("foto",file.getAbsolutePath());
    }
}

```

```

        int w = imageView.getWidth();
        int h = imageView.getHeight();
        // (*5*) Redimensiona a imagem para o tamanho do ImageView
        Bitmap bitmap = ImageResizeUtils.getResizedImage(Uri.fromFile(file), w, h, false);
        Toast.makeText(this, "w/h:" + imageView.getWidth() + "/" + imageView.getHeight()
            + " > " + "w/h:" + bitmap.getWidth() + "/" + bitmap.getHeight(),
            Toast.LENGTH_SHORT).show();
        imageView.setImageBitmap(bitmap);
    }
}
}

```

Como esse exemplo vai salvar um arquivo no SD card, é preciso declarar a permissão no arquivo de manifesto.



AndroidManifest.xml

```

<manifest . . .>
    <uses-permission android:name="android.permission.CAMERA" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <application . . . />
</manifest>

```

Também estou utilizando algumas classes da biblioteca *android-utils*, portanto declare essa dependência no arquivo *app/build.gradle*.

Ao executar essa activity, o resultado será o mesmo do anterior. A diferença é que estamos salvando a foto em um arquivo do SD card, assim podemos obter a foto com toda a resolução. Agora vamos explicar algumas partes importantes que destaquei no código-fonte.

No item (*1*) foi criado um arquivo *foto.jpg* no SD card, e esse arquivo foi passado por intent para a câmera. No emulador o arquivo foi criado na seguinte localização:

```
/storage/sdcard/Android/data/br.com.livroandroid.multimidia/files/Pictures/foto.jpg
```

Os itens (*2*) e (*3*) são referentes ao ciclo de vida da activity. Como o usuário pode girar o celular entre vertical e horizontal, precisamos salvar o estado da foto, pois conforme já explicado no livro o Android vai matar e recriar a activity nesses casos. O item (*3*) salva o arquivo no Bundle antes de destruir a activity, e o item (*2*) recupera o arquivo logo depois de recriar a activity. Dessa forma, podemos manter o estado da tela mesmo se o usuário girar o dispositivo.

O item (*4*) é o trecho de código que recebe o resultado da intent da câmera, e nesse caso basta abrir a foto do arquivo. O item (*5*) mostra o código que faz o redimensionamento da foto antes de mostrá-la no *ImageView*.

Caso o aplicativo não faça o redimensionamento da foto e o bitmap retornado pela câmera seja mostrado diretamente no `ImageView`, o Android vai apresentar erros de memória, conhecidos como **OutOfMemory**. Por isso, antes de mostrar o bitmap no `ImageView`, é feito esse redimensionamento, que consiste em diminuir o tamanho da imagem, para que ela fique somente com o tamanho necessário para ser exibida no `ImageView`. Com isso, é possível economizar recursos e memória.

Importante: entenda que o problema de memória com as fotos não está relacionado com a câmera, pois ela consegue tirar quantas fotos seu aplicativo precisar. O problema acontece caso você mostre a foto original (com toda a resolução) no `ImageView`. Portanto, nunca se esqueça de redimensionar a foto antes de mostrá-la na tela.

21.9 Trabalhando com bitmaps de forma eficiente

Existem muitas maneiras de carregar um bitmap em memória, mas tudo consiste em utilizar algum método da classe `BitmapFactory`. Por exemplo, o código a seguir carrega um bitmap em memória, a partir da `Uri` de um arquivo.

```
Bitmap bitmap = BitmapFactory.decodeFile(uriFile.getPath(), null);
```

Mas as fotos tiradas com a câmera resultam em arquivos muito grandes, devido à alta resolução das câmeras. Por exemplo, a foto de uma câmera pode ultrapassar 3.000 pixels de largura, mas muitas vezes o `ImageView` que está no layout tem apenas 300 ou 500 pixels. Nesse caso, recomenda-se redimensionar a imagem original para que ela fique com um tamanho próximo do `ImageView`.

Justamente para facilitar o redimensionamento das fotos, utilizamos a classe `ImageResizeUtils` no exemplo anterior. Essa classe mostra como redimensionar a foto da maneira correta, otimizando a memória do aplicativo. As explicações deste tópico visam fornecer a base teórica, para que você consiga entender o código-fonte dessa classe, que está disponível no GitHub.

Para redimensionar uma figura, precisamos fazer quatro passos:

1. Obter as dimensões originais de largura e altura da figura.
2. Conhecer as novas dimensões para as quais a figura deve ser redimensionada.
3. Com base nestas dimensões, é preciso calcular o fator de conversão do redimensionamento. Por exemplo, se podemos diminuir a foto na metade, ou em 1/4 do tamanho original.
4. Conhecendo o fator de conversão, basta redimensionar a foto.

No passo 1, podemos escolher qual o tamanho que a foto precisa ter. Mas o que muitos aplicativos fazem é ler o tamanho que o `ImageView` utiliza no layout. O trecho de código a seguir mostra como implementar o passo 2, a fim de extrair a largura e altura da foto, sem carregar o arquivo inteiro em memória.

```
Uri uriFile = esta é a Uri do arquivo da foto...
// Configura o BitmapFactory para apenas ler o tamanho da imagem (sem carregá-la em memória)
BitmapFactory.Options opts = new BitmapFactory.Options();
opts.inJustDecodeBounds = true;
// Faz o decode da imagem
BitmapFactory.decodeFile(uriFile.getPath(), opts);
// Lê a largura e altura do arquivo
int w = opts.outWidth; // largura original
int h = opts.outHeight // altura original
```

O segredo é utilizar o parâmetro `inJustDecodeBounds = true`. Esse parâmetro faz com que a classe `BitmapFactory` leia apenas o tamanho do arquivo, sem carregar a imagem inteira em memória. No final, as variáveis `w` e `h` contêm a largura e a altura da imagem, respectivamente.

Agora vamos para o passo 3. Uma vez que o tamanho da figura original foi obtido, podemos calcular o atributo que vai definir a escala da foto, chamado de `inSampleSize`. Esse cálculo consiste em fazer uma simples divisão do tamanho total da foto pelo tamanho desejado.

```
// Fator de escala
int scaleFactor = Math.min(w / width, h / height);
opts.inSampleSize = scaleFactor;
```

Nota: é recomendável que o atributo `inSampleSize` seja múltiplo de 2. O fator de escala funciona da seguinte forma: se o valor for igual a 1, significa que uma imagem com o tamanho original será retornada; se for igual a 2, significa metade do tamanho; se for igual a 4, significa um quarto; e assim sucessivamente.

Por último, vamos para o passo 4, que consiste em carregar o `Bitmap` em memória, mas como o fator de escala foi definido, o tamanho da foto será menor que a original. Note que para carregar a foto em memória o parâmetro `inJustDecodeBounds` deve ser `false`.

```
// Decode bitmap with inSampleSize set
options.inJustDecodeBounds = false;
Bitmap bitmap = BitmapFactory.decodeResource(res, resId, options);
```

Entender esse exemplo é fundamental para trabalhar com imagens no Android, pois mostramos uma foto em um `ImageView`, utilizando somente o tamanho e memória necessários. Note que a classe `BitmapFactory` contém vários métodos para carregar `Bitmaps` em memória, como `decodeByteArray(...)`, `decodeFile(...)`, `decodeResource(...)` etc. Qual método você vai utilizar não importa. O que você precisa fazer é utilizar corretamente os truques com o atributo `inJustDecodeBounds` para primeiro obter o tamanho original da figura, na sequência calcular o fator de escala, e por último carregar a foto reduzida em memória.

Dica: para continuar seus estudos, recomendo uma leitura nos links adicionais que estão no final do capítulo. Dentre eles, é importante ler o tópico `Loading Large Bitmaps Efficiently` da documentação oficial.

21.10 Enviando a imagem para o servidor

No mesmo exemplo em que tiramos uma foto, observe que o arquivo da foto foi salvo no SD card. Portanto, podemos transferi-lo para o servidor, caso seja necessário. Para isso, basta converter o arquivo para um array de bytes. Você pode enviar esse `byte[]` ao servidor fazendo um post HTTP, de forma que esse array seja enviado no corpo da requisição, ou fazendo um post multipart em um web service que aceite upload de arquivos.

Enfim, mostrar como implementar o servidor que aceite upload de arquivos não é o foco do livro, mas este breve tópico serve apenas para lembrá-lo que o arquivo da foto está nas suas mãos, e agora você pode fazer o que quiser com ele.

Caso precise obter mais informações a esse respeito, pesquise no Google por Android HTTP Multipart POST.

21.11 Gravando áudio e vídeo

A maneira mais simples de gravar áudio e vídeo no Android é utilizar uma intent, da mesma forma que tiramos uma foto. Essas intents são mesmo boas parceiras, não são? Aprendemos que para tirar uma foto podemos utilizar uma intent com a ação `ACTION_IMAGE_CAPTURE`, e o retorno é o arquivo da foto. Podemos utilizar o mesmo conceito para gravar vídeos, basta alterar a ação para `ACTION_VIDEO_CAPTURE`.

O código a seguir mostra como disparar a intent para gravar um vídeo:

```
File file = SDCardUtils.getPublicFile("video.mp4", Environment.DIRECTORY_MOVIES);
// Chama a intent informando o arquivo para salvar a foto
Intent i = new Intent(MediaStore.ACTION_VIDEO_CAPTURE);
i.putExtra(MediaStore.EXTRA_OUTPUT, Uri.fromFile(file));
startActivityForResult(i, 0);
```

Como resultado, o arquivo de vídeo será gravado e salvo no local informado, portanto no método `onActivityResult()` podemos mostrar o vídeo com o `VideoView`.

```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (resultCode == RESULT_OK) {
        videoView.setVideoPath(file.getAbsolutePath());
        videoView.setMediaController(new MediaController(this));
        videoView.start();
    }
}
```

Deixarei para você concluir este exercício. Caso ache necessário, veja o projeto **VideoRecorder** disponível nos exemplos do livro.

Para gravar áudio, também podemos utilizar uma intent, porém no caso do áudio o arquivo sempre é salvo no diretório interno do Android, portanto não precisamos definir o arquivo com a localização. O trecho de código a seguir mostra como disparar a intent para gravar um áudio:

```
Intent i = new Intent(MediaStore.Audio.Media.RECORD_SOUND_ACTION);
startActivityForResult(i, 0);
```

O resultado mais uma vez é entregue no método `onActivityResult()`; basta ler o método `getData()` da intent de retorno. O objeto `Uri` contém a localização do arquivo de áudio, e conforme já aprendemos esse áudio pode ser tocado com a classe `MediaPlayer`.

```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (resultCode == RESULT_OK) {
        // Uri, exemplo: content://media/external/audio/media/1
        Uri uri = data.getData();
        player = MediaPlayer.create(this, uri);
        player.start();
    }
}
```

Novamente, deixarei para você concluir este exercício, mas caso ache necessário veja o projeto **AudioRecorder** disponível nos exemplos do livro.

21.12 Links úteis

Neste capítulo, estudamos os recursos de multimídia disponíveis no Android. Para continuar seus estudos, separei alguns links interessantes.

- **Android Training – Managing Audio Playback**

<http://developer.android.com/training/managing-audio/index.html>

- **Android Training – Taking Photos Simply**

<http://developer.android.com/training/camera/photobasics.html>

- **Android Training – Recording Videos Simply**

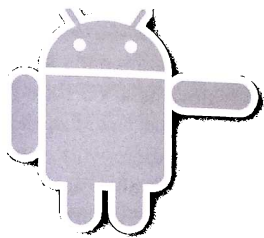
<http://developer.android.com/training/camera/videobasics.html>

- **Android Training – Controlling the Camera**

<http://developer.android.com/training/camera/cameradirect.html>

- **Android Training – Loading Large Bitmaps Efficiently**

<http://developer.android.com/training/displaying-bitmaps/load-bitmap.html>



CAPÍTULO 22

Mapas

Uma das funcionalidades que mais chamam a atenção na plataforma do Android é a grande facilidade para se construir uma aplicação integrada ao Google Maps.

Neste capítulo, veremos diversos recursos sobre a construção de mapas, como controle de zoom, tipo de visualização por rua e satélite, desenho de imagens (marcadores) no mapa e muito mais.

22.1 Introdução

Existem duas versões da API de mapas, que chamaremos de versões 1 e 2, ou apenas V1 e V2. Em dezembro de 2012, o Google descontinuou a V1. Portanto, neste livro, vamos estudar a V2. Contudo, caso seja a primeira vez que você esteja estudando a API de mapas, explicaremos alguns conceitos básicos da API antiga para você saber identificar o código deprecated (obsoleto) quando o vir.

A primeira versão da API de mapas do Android era representada pela classe `MapView`, uma subclasse de `View`.

```
<LinearLayout . . . >
    <com.google.android.maps.MapView . . . />
</LinearLayout>
```

Contudo, a API V1 e a classe `MapView` foram descontinuadas pelo Google. Dito isso, o objetivo dessa breve explicação foi apenas informar o que você não deve fazer, pois é normal que, além deste livro, você procure algum material auxiliar para complementar os seus estudos. Então, caso você encontre um artigo na internet sobre a classe `MapView`, saberá que ela foi descontinuada.

22.2 Google Maps Android API – Versão 2

Em dezembro de 2012, o Google lançou a nova API de mapas, chamada **Google Maps Android API V2**, a qual apresenta muito mais funcionalidades que a V1. Todo o framework de mapas foi criado utilizando vetores para suportar as visualizações 2D e 3D, o que também possibilita um ganho significativo no desempenho, tornando todas as interações e animações muito mais fluidas. A visualização 3D é feita automaticamente se o nível de zoom estiver perto o bastante e se na cidade em questão houver imagens 3D.

A API também ganhou diversas melhorias, mas a principal foi a utilização de fragments. Antigamente, com a V1, só era possível exibir um mapa de cada vez na tela, o que, no caso dos tablets, trouxe um grande problema. Mas como a V2 é implementada com fragments, essa limitação não existe mais. Na prática, trocamos a classe `MapView` por `MapFragment`. Caso esteja utilizando a biblioteca de compatibilidade, que é o nosso caso no aplicativo dos carros, use a classe `SupportMapFragment`.

22.3 Google Play Services

O Google Play Services é um aplicativo distribuído no Google Play que apresenta funcionalidades essenciais para se comunicar com os serviços do Google, além de várias APIs. Por exemplo, a API de Mapas V2 faz parte do Google Play Services, assim como diversas outras APIs, tais como: Google+, Google Drive, Google Fit, Jogos, Localização etc.

A vantagem do Google Play Services é que ele é distribuído pela loja de aplicativos; sendo assim, pode receber atualizações, e qualquer melhoria ou correção de bug das APIs do Google são atualizadas facilmente nos dispositivos. Para utilizar o Google Play Services no projeto, basta declarar a dependência no arquivo *app/build.gradle*. Faça isso no projeto dos carros, pois neste capítulo vamos desenvolver a tela que vai mostrar o mapa da fábrica do carro.



`app/build.gradle`

...

```
compile 'com.google.android.gms:play-services:7.0.0'
```

Caso o Android Studio informe que existe uma versão mais nova da biblioteca quando você for estudar o livro, fique à vontade para utilizar sempre as novas versões.

Nota: o Google está criando ou movendo algumas APIs para o Google Play Services. A vantagem é que melhorias ou bugs podem ser corrigidos atualizando o aplicativo do Google Play Services pela loja.

22.4 Gerando a chave de acesso dos mapas

Para o mapa funcionar, precisamos criar uma chave de autenticação no serviço do Google. A criação das chaves é feita no Google Developers Console, que pode ser acessado na seguinte página:

<https://console.developers.google.com/>

Nota: para fazer login na página do Google Developers Console é necessária uma conta do Google, ou seja, do Gmail.

Se for a primeira vez que estiver abrindo essa página com sua conta de email, você verá uma tela com um botão **Create project**, utilizado para criar um novo projeto. No formulário de criação do projeto, basta digitar um nome qualquer para identificá-lo. Eu recomendo que você crie um projeto chamado **Livro Android**, assim todas as configurações que fizermos referente ao livro ficarão nele.

Ao criar o projeto, note que o campo **Project ID** é preenchido automaticamente, e você não precisa se preocupar com ele.

Nota: um projeto no console do Google representa um conjunto de configurações que são feitas na sua conta para que as aplicações utilizem determinados serviços, como o de mapas, mensagens de push, entre outros.

Logo depois de criar a configuração do projeto, você verá uma página com algumas opções no menu lateral, como: **Overview**, **APIs & auth**, **Monitoring**, **Compute**, **Networking**, **Storage**, **Big Data** etc.

A primeira configuração que temos de fazer na página do console é habilitar o serviço dos mapas. Para isso, acesse a página **APIs & auth** > **APIs** e habilite o serviço **Google Maps Android API v2**, conforme a figura 22.1. A figura mostra minha conta de testes. Veja que o serviço **Google Cloud Messaging for Android** para enviar mensagens por push também está habilitado, mas vamos estudar esse assunto em outro capítulo.

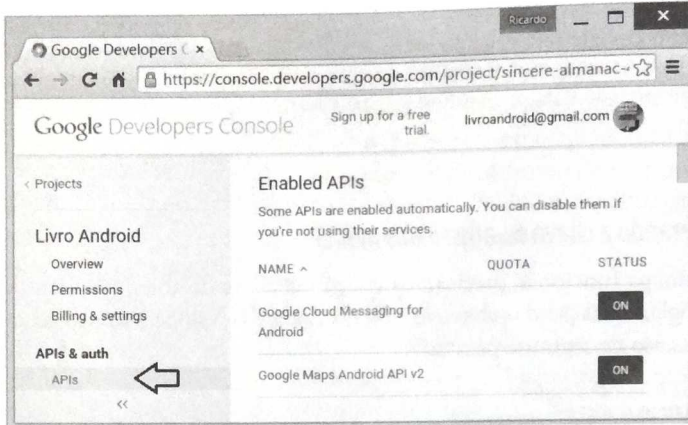


Figura 22.1 – Google Maps Android API v2 habilitado.

O próximo passo é criar a chave de acesso para o serviço do Google Maps. Para isso, precisamos obter o **SHA-1 fingerprint** do certificado que você utiliza para compilar o projeto. Por padrão, um certificado de debug é criado automaticamente pelo Android Studio, o qual fica na pasta do usuário do sistema operacional. A seguir, temos a localização deste arquivo no Windows e Linux:

- **Windows** – `C:\Users\usuario\.android\debug.keystore`
- **Linux** – `/home/usuario/.android/debug.keystore`

Esse é o certificado de debug utilizado para testes, a fim de instalar aplicativos no emulador ou em um dispositivo. Quando você for publicar o aplicativo no Google Play, precisará criar outro certificado, mas isso vamos estudar no capítulo sobre o Gradle.

Nota: um aplicativo sempre precisa ser assinado por um certificado antes de ser instalado no dispositivo. O Android Studio faz isso automaticamente ao executar o projeto, assinando-o com o certificado de debug, que é o arquivo *debug.keystore* localizado na pasta do usuário.

Copie o caminho de seu certificado, abra um prompt e digite o seguinte comando para extrair o **SHA-1 fingerprint**. Note que informei o caminho do arquivo *debug.keystore*.

```
C:\keytool -list -v -keystore "C:\Users\usuario\.android\debug.keystore"
-alias androiddebugkey -storepass android -keypass android
Alias name: androiddebugkey
```

Certificate fingerprints:

MD5: 14:A0:36:6C:7C:50:E6:42:88:8D:94:2F:C4:CF:03:95

SHA1: C1:66:56:93:DF:DE:0B:B9:DC:ED:76:D7:65:B7:10:DC:1F:3F:0D:41

Signature algorithm name: SHA1withRSA

Version: 3

O resultado do comando mostra várias informações sobre o certificado. Mas a linha pela qual estamos interessados é esta que exhibe o **SHA1 fingerprint**:

SHA1: C1:66:56:93:DF:DE:0B:B9:DC:ED:76:D7:65:B7:10:DC:1F:3F:0D:41

Nota: o comando `keytool` é um executável que está localizado na pasta `bin` do JDK.

Depois de copiar o valor do **SHA-1 fingerprint**, volte para a página do Google Developers Console. Entre na página **APIs & auth > Credentials** e clique no botão **Create New Key**. Na sequência, selecione o tipo **Android Key**. Na janela que vai aparecer (Figura 22.2), digite no campo de texto o SHA-1 fingerprint + ponto e vírgula + o pacote do projeto, como por exemplo:

C1:66:56:93:DF:DE:0B:B9:DC:ED:76:D7:65:B7:10:DC:1F:3F:0D:41;br.com.livroandroid.carros

A figura 22.2 mostra como digitar o fingerprint com o pacote.

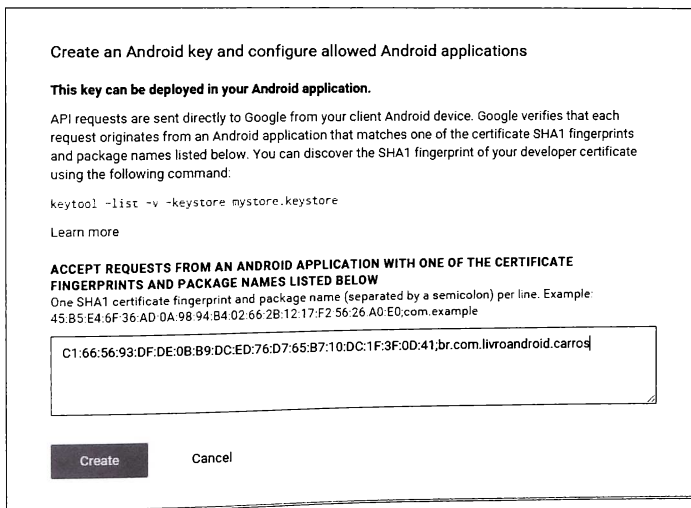


Figura 22.2 – Criando uma chave de acesso Android Key.

O resultado disso é que foi criada uma chave de acesso ao **Google Maps Android API v2** para o seu certificado e pacote. Isso significa que somente o seu projeto poderá utilizá-la. A figura 22.3 mostra a página com a chave criada. O campo **API Key** é a chave de que precisamos, que neste caso é o texto `AIzaSyAap0vUMdKvqd05bqZbp2Sy4hqXmSQU_zk`.

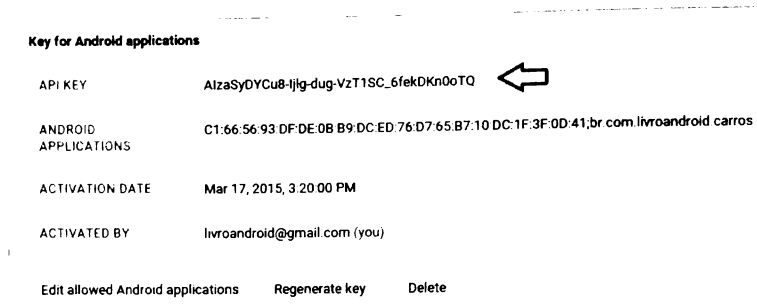


Figura 22.3 – Chave de acesso criada.

Nota: uma chave de acesso é utilizada por uma aplicação para autenticar o acesso nos serviços do Google. Copie a **API Key**, pois vamos utilizá-la depois para configurar o arquivo de manifesto do projeto.

22.5 Configurando o projeto

Vamos continuar com o desenvolvimento do projeto dos carros, mas antes de escrever código com a API V2 precisamos configurar o projeto. Como dito antes, a API V2 de mapas depende do Google Play Services, portanto certifique-se de que essa dependência está configurada no `app/build.gradle`.



`app/build.gradle`

...

```
compile 'com.google.android.gms:play-services:7.0.0'
```

A parte mais trabalhosa é configurar o arquivo de manifesto, então adicione as seguintes permissões e configurações:



`AndroidManifest.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... package="br.com.livroandroid.carros">
    <uses-permission android:name="android.permission.INTERNET" />
```

```

<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
<!-- GPS por hardware -->
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<!-- Obter a localização por Wi-Fi ou triangulação de antenas -->
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<!-- Mapas V2 depende da OpenGL ES V2 -->
<uses-feature android:glEsVersion="0x00020000" android:required="true"/>
<application . . .>
    <!-- Google Play Services -->
    <meta-data
        android:name="com.google.android.gms.version"
        android:value="@integer/google_play_services_version" />
    <!-- Chave de acesso (API Key) criada na página de Console. -->
    <meta-data android:name="com.google.android.maps.v2.API_KEY"
        android:value="@string/API_KEY"/>
    <!-- Continua com todas as activities aqui -->
    <activity . . . />
</application>
</manifest>

```

No arquivo de manifesto, estamos usando o texto `@string/API_KEY`, então vamos criar o arquivo `/res/values/strings_config.xml` com essa string de configuração. Note que não estou usando o arquivo `strings.xml` padrão, pois gosto de separar essas configurações em um arquivo isolado. Em outros capítulos, também vamos adicionar mais configurações nesse arquivo.

 `/res/values/strings_config.xml`

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- Gerado do SHA1: C1:66:56:93:DF:DE:0B:B9:DC:ED:76:D7:65:B7:10:DC:1F:3F:0D:41 -->
    <string name="API_KEY">AIzaSyDYCu8-Ijlg-dug-VzT1SC_6fekDKn0tQ</string>
</resources>

```

Agora, vamos entender esse arquivo. Primeiramente, adicionamos algumas permissões que são necessárias para os mapas funcionarem. Na lista a seguir, vamos aproveitar para revisar o significado de cada permissão que temos até o momento no aplicativo dos carros.

```
<uses-permission android:name="android.permission.INTERNET" />
```

Utilizada para acessar a internet.

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Utilizada para ler o estado da rede.

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
```

Utilizada para o Google Maps salvar informações no SD card, como por exemplo: fazer cache dos mapas.

```
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Utilizada para ler dados do SD card.

```
<uses-permission android:name="com.google.android.providers.gsf.permission.READ_GSERVICES" />
```

Utilizada para acessar os serviços do Google.

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
```

Utilizada para acessar o GPS por triangulação de antenas.

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
```

Utilizada para acessar o GPS por hardware.

Nota: mesmo que você não utilize a API de localização no código, caso a opção `my location` esteja habilitada no mapa pela API, as permissões de GPS são necessárias.

Logo depois das permissões, outra configuração importante é declarar que a OpenGL ES V2 é requisito para os mapas funcionarem. Sendo assim, precisamos adicionar a configuração `<uses-feature>` para que somente dispositivos com a OpenGL ES V2 possam instalar essa aplicação pelo Google Play:

```
<!-- Precisa de OpenGL ES versão 2 -->  
<uses-feature android:glEsVersion="0x00020000" android:required="true" />
```

Esta página mostra informações sobre a versão da Open GL encontrada nos diversos dispositivos com Android:

<http://developer.android.com/about/dashboards/index.html#OpenGL>

Outra configuração importante é declarar a versão da biblioteca do Google Play Services com o qual o seu projeto foi compilado.

```
<!-- Versão do Google Play Services -->  
<meta-data android:name="com.google.android.gms.version"  
    android:value="@integer/google_play_services_version" />
```

Por último, a configuração mais importante de todas é declarar a chave de acesso ao serviço de mapas. Lembra aquela chave (API Key) que criamos na página do Google Developers Console? Coloque a sua chave aqui.

```
<!-- Chave de acesso ao Google Play services -->
<meta-data android:name="com.google.android.maps.v2.API_KEY"
    android:value="@string/API_KEY" />
```

Lembrando que o texto @string/API_KEY eu gosto de cadastrar no arquivo */res/values/strings_config.xml* para separar textos do aplicativo de strings de configuração.

 */res/values/strings_config.xml*

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <!-- Gerado do SHA1: C1:66:56:93:DF:DE:0B:B9:DC:ED:76:D7:65:B7:10:DC:1F:3F:0D:41 -->
    <string name="API_KEY">AIzaSyDYCu8-Ijlg-dug-VzT1SC_6fekDKn0oTQ</string>
</resources>
```

Importante: para testar os mapas, utilize um dispositivo real com o Google Play Services instalado, pois o emulador não tem esse pacote. Embora existam artifícios técnicos para instalá-lo no emulador, não abordarei isso aqui. Lembre-se também de criar a sua chave API Key na página de console do Google, pois ela será gerada para o certificado que você está utilizando no seu computador. Se você não inserir a chave correta, o mapa não vai funcionar.

Lembre-se de que cada chave é diferente e você precisa utilizar a sua, caso contrário o mapa não vai funcionar. Isso porque cada chave é gerada baseada no certificado digital utilizado no desenvolvimento, e naturalmente o arquivo *debug.keystore* que está na sua máquina é diferente do meu.

Uma dica interessante no caso de empresas nas quais existem vários desenvolvedores no time é utilizar o mesmo arquivo *debug.keystore* em todos os computadores. Assim a chave para todos é a mesma, então você pode instalar o aplicativo de seu computador, ou do computador de um colega de trabalho, pois todos utilizam o mesmo arquivo *debug.keystore* para assinar o aplicativo. Lembrando que todas as aplicações no Android precisam ser assinadas por um certificado e, caso já exista uma aplicação instalada no dispositivo, somente uma aplicação assinada com o mesmo certificado pode atualizá-la.

22.6 Adicionando o mapa no projeto dos carros

Depois que configuramos todo o projeto, vamos para a parte mais fácil, que é mostrar a fábrica do carro no mapa. No XML ou JSON dos carros, existem os campos com a latitude e longitude da fábrica de cada carro.

```
{
    "nome": "Carro 1",
    . . .
    "latitude": "latitude aqui",
    "longitude": "longitude aqui"
}
```

Por exemplo, o carro Ferrari FF está com a localização definida para a fábrica da Ferrari, em algum lugar da Itália. Nosso objetivo é mostrar uma activity com um mapa posicionado na localização da fábrica do carro. Faremos isso ao clicar no botão **Mapa** que já existe na tela de detalhes do carro.

O passo a passo para mostrar um mapa será parecido com o que fizemos para mostrar um vídeo. No caso do vídeo, criamos as classes `VideoActivity` para controlar a navegação, e criamos a classe `VideoFragment` para gerenciar o conteúdo e o layout. Dentro do `VideoFragment` utilizamos o `VideoView`. Desta vez, vamos criar as classes `MapaActivity` e `MapaFragment`. Dentro da classe `MapaFragment`, vamos inserir o componente de mapas da API V2, que é o `fragment SupportMapFragment`. Então mãos à obra! Abra o projeto dos carros novamente e crie uma activity conforme demonstrado a seguir. Ela recebe o objeto carro por parâmetro e delega o trabalho para um fragment.



MapaActivity.java

```
package br.com.livroandroid.carros.activity;
. . .
import br.com.livroandroid.carros.fragments.MapaFragment;
public class MapaActivity extends BaseActivity {
    private Carro carro;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_mapa);
        // Configura a Toolbar como a action bar
        setUpToolbar();
        carro = (Carro) getIntent().getSerializableExtra("carro");
        getSupportActionBar().setTitle(carro.nome);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        if (savedInstanceState == null) {
```

```

        // Adiciona o fragment no layout da activity
        MapaFragment mapaFragment = new MapaFragment();
        mapaFragment.setArguments(getIntent().getExtras());
        getSupportFragmentManager().beginTransaction().replace(R.id.fragLayout,
            mapaFragment).commit();
    }
}
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Volta para a activity CarroActivity
    // Mesmo código que colocamos na classe VideoActivity
}
}

```

No arquivo de layout, vamos colocar a Toolbar e um `FrameLayout` de marcação para adicionar o fragment.

 `/res/layout/activity_mapa.xml`

```

<LinearLayout android:orientation="vertical"
    android:layout_width="match_parent" android:layout_height="match_parent">
    <include layout="@layout/include_toolbar" />
    <FrameLayout android:id="@+id/fragLayout"
        android:layout_width="match_parent" android:layout_height="match_parent"
        tools:layout="@layout/fragment_mapa" />
</LinearLayout>

```

Por último, configure a activity no arquivo de manifesto.

 `AndroidManifest.xml`

```

<application . . .>
    . . .
    <activity android:name=".activity.MapaActivity"
        android:parentActivityName=".activity.CarroActivity" >
        <meta-data android:name="android.support.PARENT_ACTIVITY"
            android:value=".activity.CarroActivity" />
    </activity>
</application>

```

O conteúdo da tela será gerenciado pelo fragment, portanto crie a classe `MapaFragment`. O código a seguir mostra um template básico de como utilizar o fragment `SupportMapFragment`, pois estamos posicionando o mapa na coordenada da fábrica, mostrando um marcador e configurando o tipo do mapa (normal, satélite, terreno ou híbrido).



MapaFragment.java

```
package br.com.livroandroid.carros.fragments;
...
import com.google.android.gms.maps.SupportMapFragment;
public class MapaFragment extends BaseFragment implements OnMapReadyCallback {
    // Objeto que controla o Google Maps
    private GoogleMap map;
    private Carro carro;
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_mapa, container, false);
        // Recupera o fragment que está no layout
        // Utiliza o getChildFragmentManager() pois é um fragment dentro do outro
        SupportMapFragment mapFragment = (SupportMapFragment)
            getChildFragmentManager().findFragmentById(R.id.mapFragment);
        // Inicia o Google Maps dentro do fragment
        mapFragment.getMapAsync(this);
        this.carro = (Carro) getArguments().getSerializable("carro");
        return view;
    }
    @Override
    public void onMapReady(GoogleMap map) {
        // O método onMapReady(map) é chamado quando a inicialização do mapa estiver Ok.
        this.map = map;
        if(carro != null) {
            // Ativa o botão para mostrar minha localização
            map.setMyLocationEnabled(true);
            // Cria o objeto LatLng com a coordenada da fábrica
            LatLng location = new LatLng(Double.parseDouble(carro.latitude),
                Double.parseDouble(carro.longitude));
            // Posiciona o mapa na coordenada da fábrica (zoom = 13)
            CameraUpdate update = CameraUpdateFactory.newLatLngZoom(location, 13);
            map.moveCamera(update);
            // Marcador no local da fábrica
            map.addMarker(new MarkerOptions()
                .title(carro.nome)
                .snippet(carro.desc)
                .position(location));
            // Tipo do mapa:
            // (normal, satélite, terreno ou híbrido)
            map.setMapType(GoogleMap.MAP_TYPE_NORMAL);
        }
    }
}
```

Importante: como o fragment da biblioteca de mapas `SupportMapFragment` está dentro do nosso fragment `MapaFragment`, temos uma situação que ainda não tínhamos visto no livro, que é um fragment dentro de outro. Por isso, para obter o fragment `SupportMapFragment` foi utilizado o método `getChildFragmentManager()` em vez de `getFragmentManager()`.

O arquivo de layout do fragment contém apenas o fragment `SupportMapFragment`.

```
 /res/layout/fragment_video.xml

<FrameLayout ... >
    <fragment android:id="@+id/mapFragment"
        class="com.google.android.gms.maps.SupportMapFragment"
        android:layout_width="match_parent" android:layout_height="match_parent"
    />
</FrameLayout>
```

Para finalizar o exemplo, volte à classe `CarroFragment`, que é a tela com os detalhes do carro que tem os botões **Video** e **Mapa** na action bar. Altere o trecho de código do `PopupMenu` cujo alerta inflamos com as opções. Na opção que mostra o mapa, vamos navegar para a activity que acabamos de criar.

```
 CarroFragment.java

public class CarroFragment extends BaseFragment {
    ...
    popupMenu.setOnMenuItemClickListener(new PopupMenu.OnMenuItemClickListener() {
        @Override
        public boolean onMenuItemClick(MenuItem item) {
            ...
            else if (item.getItemId() == R.id.action_mapa) {
                // Abre outra activity para mostrar o mapa
                Intent intent = new Intent(getContext(), MapaActivity.class);
                intent.putExtra("carro", carro);
                startActivity(intent);
            }
            return true;
        }
    });
}
```

Pronto! A navegação de telas está concluída, então execute o projeto no emulador para testar. Na tela de detalhes do carro, ao selecionar a opção **Mapa**, vamos navegar para a activity do mapa, conforme a figura 22.4. O resultado é a tela do mapa

posicionado na fábrica do carro. Foi adicionado um marcador no local da fábrica, que mostra informações sobre essa localização. Observe que o mapa tem o botão **my location** habilitado, que ao ser clicado vai posicionar o mapa na localização do usuário. Nos tópicos a seguir, vamos estudar mais detalhes da API.

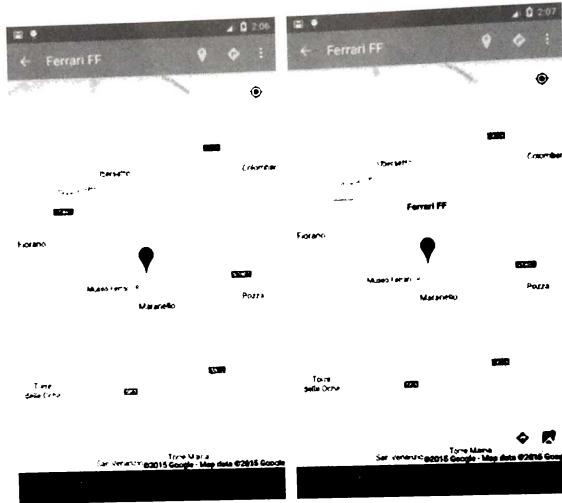


Figura 22.4 – Mapa posicionado na fábrica do carro.

22.7 Classe GoogleMap

Depois de colocar o fragment `SupportMapFragment` no layout, podemos recuperá-lo normalmente com o método `findViewById(id)`. Precisamos fazer isso para configurar a visualização do mapa.

Na verdade, o fragment `SupportMapFragment` não faz nada, ele apenas cria a view necessária para o mapa funcionar. Mas dentro desse fragment existe o objeto `com.google.android.gms.maps.GoogleMap`. Agora, falando em português bem claro, é esse o cara! A classe `GoogleMap` representa o mapa do Google e, por meio dela, podemos controlar visualização do mapa, nível de zoom, aparência, localização, inserir marcadores etc. Essa classe faz um trabalho importante e encapsula toda a complexidade de acesso aos mapas do Google, tornando o trabalho do desenvolvedor mais simples.

Para obter o objeto `GoogleMap`, é necessário se conectar aos serviços do Google. Isso é feito com o método `getMapAsync(listener)`. Esse método é assíncrono, e o resultado será entregue no método `onMapReady(map)` da interface `OnMapReadyCallback`

que você precisa implementar. Uma vez que temos o objeto `GoogleMap` em mãos, é possível brincar com o mapa, como alterar sua posição, adicionar marcadores, alterar o tipo de visualização, fazer desenhos etc. Para revisar os conceitos, veja o código-fonte da classe `MapaFragment` que fizemos.

22.8 Localização do mapa – latitude e longitude

Na API de mapas, uma coordenada é representada pela classe `LatLng`, que pode ser criada informando a latitude e longitude:

```
LatLng latLng = new LatLng(-23.564224, -46.653156);
```

Uma vez que a localização é conhecida, podemos criar um objeto do tipo `CameraUpdate` para posicionar o mapa no local desejado, conforme demonstrado neste trecho de código:

```
GoogleMap map = ...;
// Localização do mapa (latitude e longitude)
LatLng latLng = new LatLng(-23.564224, -46.653156);
CameraUpdate position = CameraUpdateFactory.newLatLngZoom(latLng, 15); // O número 15 é
                                                                    // o zoom
map.moveCamera(position);
```

Nesse código, utilizei o método `moveCamera(update)` para alterar a localização do mapa, porém ele faz isso de forma brusca (sem animação). Caso seja necessário fazer essa atualização com uma animação, utilize o método `animateCamera(update)`, o que torna a experiência do usuário bem mais amigável e fluida. O método `animateCamera(update, time, callback)` contém uma variação com três argumentos que basicamente definem o tempo em milissegundos que a animação deve durar e um listener de callback caso a animação seja cancelada. Apenas para demonstrar a utilização da API, o trecho de código a seguir mostra como centralizar o mapa em determinada localização de forma animada, com duração de dez segundos. O efeito criado é semelhante à abertura do Google Earth, no qual você entra no globo terrestre de forma animada.

```
// Centraliza o mapa com animação de dez segundos
CameraUpdate update = CameraUpdateFactory.newLatLngZoom(location, 13);
map.animateCamera(update, 10000, new CancelableCallback() {
    @Override
    public void onFinish() {
        Toast.makeText(getContext(), "Mapa centralizado.", Toast.LENGTH_SHORT).show();
    }
    @Override
```

```
public void onCancel() {  
    Toast.makeText(getContext(), "Animação cancelada.", Toast.LENGTH_SHORT).show();  
}  
});
```

22.9 CameraPosition – zoom

Como você pôde perceber nos exemplos anteriores, foi configurada a localização do mapa, assim como o zoom. Fizemos isso com o método `CameraUpdateFactory.newLatLngZoom(latLng, zoom)`, que também recebe o zoom como parâmetro:

```
GoogleMap map = ...;  
LatLng latLng = new LatLng(-23.564224, -46.653156);  
int zoom = 15;  
CameraUpdate update = CameraUpdateFactory.newLatLngZoom(latLng, zoom);  
map.moveCamera(update);
```

Outra forma de configurar os valores do mapa, como localização e zoom, é utilizar a classe `CameraPosition.Builder`:

```
GoogleMap map = ...;  
LatLng latLng = new LatLng(-23.564224, -46.653156);  
CameraPosition position = new CameraPosition.Builder().target(latLng).zoom(15).build();  
CameraUpdate update = CameraUpdateFactory.newCameraPosition(position);  
map.moveCamera(update);
```

Note que, para alterar a posição e o zoom da câmera, é necessário criar primeiramente um objeto `CameraPosition`, o qual pode facilmente ser criado pelo `CameraPosition.Builder`, que, como o próprio nome indica, é uma classe de criação de objetos que segue o padrão `Builder` do GoF. Depois que o objeto `CameraPosition` for criado, é necessário criar o `CameraUpdate` baseado nele, e somente depois disso atualizam-se essas informações no mapa com os métodos `moveCamera(update)` ou `animateCamera(update)`.

Resumindo, este trecho de código que utilizamos anteriormente:

```
CameraUpdate update = CameraUpdateFactory.newLatLngZoom(latLng, 15); // 15 é o zoom
```

é um atalho para este código:

```
CameraPosition position = new CameraPosition.Builder().target(latLng).zoom(15).build();  
CameraUpdate update = CameraUpdateFactory.newCameraPosition(position);
```

Mas a classe `CameraUpdateFactory` tem vários atalhos. Por exemplo, se você precisa configurar apenas o zoom, basta utilizar o seguinte código:

```
CameraUpdate update = CameraUpdateFactory.zoomTo(10);
```

A vantagem de utilizar a classe `CameraUpdateFactory` para atualizar o zoom é que ela vai manter a localização atual e apenas alterar o zoom. Outra forma de controlar o zoom é utilizar os seguintes métodos:

`CameraUpdateFactory.zoomIn()`

Aumenta o zoom do mapa em 1.

`CameraUpdateFactory.zoomOut()`

Diminui o zoom do mapa em 1.

`CameraUpdateFactory.zoomTo(zoom)`

Configura o valor do zoom utilizado pelo mapa.

`CameraUpdateFactory.zoomBy(zoom)`

Aumenta ou diminui o zoom do mapa, conforme o valor informado.

Nota: os valores válidos para o zoom vão de 2 a 22.

22.10 Configurando o tipo do mapa

Usuários dos mapas do Google na internet já estão acostumados com os modos de visualização dos mapas. No Android, essas opções também estão presentes.

O tipo do mapa é configurado pelo método `setMapType(tipo)` da classe `GoogleMap`, o qual recebe uma das seguintes constantes:

`GoogleMap.MAP_TYPE_NONE`

Modo de visualização mais simples do mapa, de forma que nenhuma informação extra é exibida.

`GoogleMap.MAP_TYPE_NORMAL`

Modo de visualização-padrão dos mapas, no qual podemos visualizar as ruas, estradas e rios.

`GoogleMap.MAP_TYPE_SATELLITE`

Modo de visualização com os dados do satélite.

GoogleMap.MAP_TYPE_HYBRID

Modo de visualização com os dados fotográficos do satélite, com os mapas das ruas. É o modo de satélite mais detalhado.

GoogleMap.MAP_TYPE_TERRAIN

Modo de visualização que exhibe os dados topográficos do mapa.

22.11 Colocando os conceitos em práticas

Depois de aprendermos mais detalhes sobre a API de mapas, como controlar o zoom e alterar o tipo de visualização, vamos escrever um pouco de código para deixar a tela de mapa mais interessante.

Crie o seguinte arquivo de menu para o fragment dos mapas:

 /res/menu/menu_frag_mapa.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <!-- Localização da fábrica do carro -->
    <item
        android:id="@+id/action_location_carro"
        android:title="@string/mapa_location_carro"
        android:icon="@drawable/ic_action_place"
        app:showAsAction="always" />
    <!-- Traçar rota -->
    <item
        android:id="@+id/action_location_directions"
        android:title="@string/mapa_location_directions"
        android:icon="@drawable/ic_action_directions"
        app:showAsAction="always" />
    <!-- Modo normal -->
    <item
        android:id="@+id/action_mapa_normal"
        android:title="@string/mapa_normal"
        app:showAsAction="never" />
    <!-- Modo satélite -->
    <item
        android:id="@+id/action_mapa_satelite"
        android:title="@string/mapa_satelite"
        app:showAsAction="never" />
```

```

<!-- Modo terreno -->
<item
    android:id="@+id/action_mapa_terreno"
    android:title="@string/mapa_terreno"
    app:showAsAction="never" />
<!-- Modo hibrido -->
<item
    android:id="@+id/action_mapa_hibrido"
    android:title="@string/mapa_hibrido"
    app:showAsAction="never" />
<!-- Zoom in (+) -->
<item
    android:id="@+id/action_zoom_in"
    android:title="@string/mapa_zoom_in"
    app:showAsAction="never" />
<!-- Zoom out (-) -->
<item
    android:id="@+id/action_zoom_out"
    android:title="@string/mapa_zoom_out"
    app:showAsAction="never" />
</menu>

```

Esse arquivo de menu contém opções para alterar o modo de visualização do mapa (normal, satélite, terreno e híbrido) e ações para aumentar ou diminuir o zoom. Também adicionei uma ação para mostrar a localização da fábrica do carro e outra para mostrar uma rota da localização atual até a localização da fábrica do carro.

Para o código compilar, crie estas strings:

```

 /res/values/strings.xml

<?xml version="1.0" encoding="utf-8"?>
<resources>
    . . .
    <!-- menu_frag_mapa.xml -->
    <string name="mapa_normal">Modo Normal</string>
    <string name="mapa_satelite">Modo Satélite</string>
    <string name="mapa_terreno">Modo Terreno</string>
    <string name="mapa_hibrido">Modo Híbrido</string>
    <string name="mapa_location_directions">Direções</string>
    <string name="mapa_location_carro">Localização do Carro</string>
    <string name="mapa_zoom_in">Zoom (+)</string>
    <string name="mapa_zoom_out">Zoom (-)</string>
</resources>

```

Feito isso, basta inflarmos esse menu para mostrar os itens na action bar. O código está comentado, portanto leia com atenção. Lembre-se de que para um fragment adicionar ações na action bar ele precisa chamar o método `setHasOptionsMenu(true)`.



MapaFragment.java

```
...
public class MapaFragment extends BaseFragment {
    ...
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_mapa, container, false);
        setHasOptionsMenu(true);
        ...
        return view;
    }
    ...
    @Override
    public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
        super.onCreateOptionsMenu(menu, inflater);
        inflater.inflate(R.menu.menu_frag_mapa, menu);
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        if(map != null && carro != null) {
            if (item.getItemId() == R.id.action_location_carro) {
                // Posiciona mapa na localização da fábrica
                LatLng location = new LatLng(Double.parseDouble(carro.latitude),
                    Double.parseDouble(carro.longitude));
                map.animateCamera(CameraUpdateFactory.newLatLngZoom(location, 13));
            } else if (item.getItemId() == R.id.action_location_directions) {
                // Posiciona mapa no usuário
                toast("Mostrar rota/direções até a fábrica.");
            } else if (item.getItemId() == R.id.action_zoom_in) {
                toast("zoom +");
                map.animateCamera(CameraUpdateFactory.zoomIn());
            } else if (item.getItemId() == R.id.action_zoom_out) {
                toast("zoom -");
                map.animateCamera(CameraUpdateFactory.zoomOut());
            } else if (item.getItemId() == R.id.action_mapa_normal) {
                // Modo Normal
                map.setMapType(GoogleMap.MAP_TYPE_NORMAL);
            } else if (item.getItemId() == R.id.action_mapa_satelite) {
```

```

        // Modo Satélite
        map.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
    } else if (item.getItemId() == R.id.action_mapa_terreno) {
        // Modo Terreno
        map.setMapType(GoogleMap.MAP_TYPE_TERRAIN);
    } else if (item.getItemId() == R.id.action_mapa_hibrido) {
        // Modo Híbrido
        map.setMapType(GoogleMap.MAP_TYPE_HYBRID);
    }
}
return super.onOptionsItemSelected(item);
}
}

```

Desta vez, ao executar o projeto no emulador, teremos as opções para trocar o modo de visualização do mapa. A figura 22.5 mostra o mapa nos modos Normal, Satélite, Terreno e Híbrido. Também adicionei os botões **Zoom (+)** e **Zoom (-)**, que brincam com o zoom do mapa, basta você testar. Todas essas opções ficam no menu overflow da action bar. Como action buttons deixei apenas o botão que vai posicionar o mapa na coordenada do carro e o botão para traçar uma rota (falaremos da rota posteriormente).

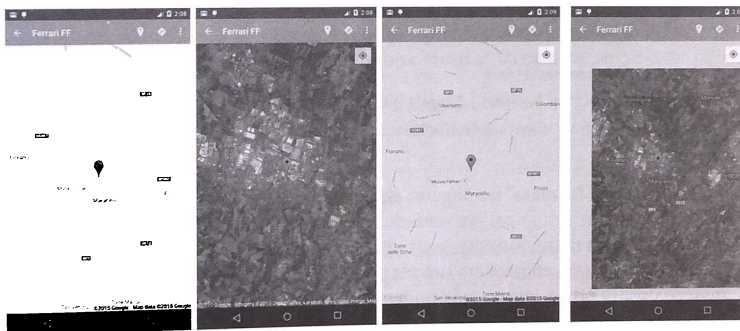


Figura 22.5 – Modos de visualização do mapa.

22.12 CameraPosition

Conforme já estudamos, a classe `CameraPosition` representa a posição do mapa e, nela, podemos configurar tanto a localização como também o zoom. Igualmente vimos que a classe utilitária `CameraUpdateFactory` pode facilitar o trabalho, pois

realiza dois passos em um só. Contudo, a classe `CameraPosition` faz muito mais do que apenas informar a localização e o zoom, sendo assim, vamos explorar alguns métodos importantes como o `bearing(graus)` e `tilt(graus)`.

Vamos continuar os estudos sobre a API de mapas, mas faremos isso fora do projeto dos carros. Para acompanhar as próximas explicações, abra o projeto deste capítulo **Demo-MapsV2** no Android Studio. O trecho de código a seguir demonstra os métodos `bearing(graus)` e `tilt(graus)`, que também podemos informar para turbinar a visualização do mapa:

```
GoogleMap map = ...;
// Localização do mapa (Av. Paulista - SP)
LatLng latLng = new LatLng(-23.564224, -46.653156);
final CameraPosition position = new CameraPosition.Builder()
    .target(latLng) // Localização
    .bearing(0)     // Rotação da câmera em graus
    .tilt(0)        // Ângulo em que a câmera está posicionada em graus
    .zoom(15)       // Zoom
    .build();

CameraUpdate update = CameraUpdateFactory.newCameraPosition(position);
```

Note que a linguagem Java permite quebrar a linha depois do ponto, para chamar múltiplos métodos em sequência, mas isso também pode ser escrito em uma única linha de código, conforme a sua preferência:

```
final CameraPosition position =
    new CameraPosition.Builder().target(latLng).bearing(0).tilt(0).zoom(15).build();
CameraUpdate update = CameraUpdateFactory.newCameraPosition(position);
```

Nota: a palavra “câmera” pode gerar alguma confusão caso você a associe com uma câmera que tira fotos. Na verdade, o termo “câmera” é utilizado porque a visualização do mapa funciona exatamente como se estivéssemos enxergando o mundo com uma câmera em um plano linear. Por padrão, a câmera exibe os mapas de cima para baixo, mas podemos alterar esse ângulo de visualização, rotação da câmera etc.

22.13 CameraPosition – bearing “rotação”

Um dos parâmetros mais interessantes da classe `CameraPosition` é o `bearing`, que configura a orientação que a câmera está apontando.

Para entendermos melhor, vamos verificar a figura 22.6, que exhibe o mapa posicionado no Brasil e com um zoom = 0. Note que as figuras do centro e da direita foram rotacionadas em 45° e 90°, respectivamente. Esse comportamento é justamente o que faz a propriedade `bearing` da classe `CameraPosition`.

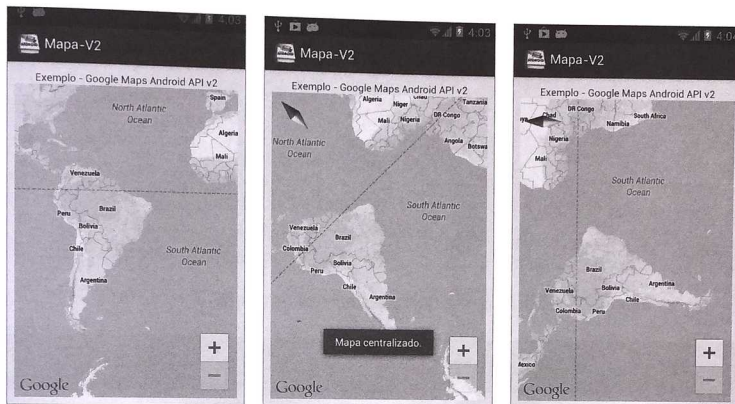


Figura 22.6 – Parâmetro `bearing` (rotação) com os valores 0°, 45° e 90°.

Nota: para rotacionar o mapa manualmente pelo touch, utilize dois dedos e faça um círculo como se estivesse girando uma moeda. Na prática, isso é o mesmo que alterar a propriedade `bearing`.

22.14 `CameraPosition` – tilt “inclinação”

Outro parâmetro muito interessante da classe `CameraPosition` é o `tilt`, que configura a inclinação da câmera. A figura 22.7 exhibe o mapa na Av. Paulista – SP, estando as figuras do centro e da direita com uma inclinação (`tilt`) de 45° e 90° respectivamente.

O efeito de inclinação fica ainda mais claro quando o mapa está com um zoom maior. Por exemplo, a partir do zoom = 17, algumas cidades já exibem visualizações em 3D. A figura 22.8 mostra o mesmo endereço com um zoom maior. Nela, é possível visualizar os mapas em 3D. Nesse exemplo, podemos visualizar claramente como a inclinação afeta a visualização do mapa. A visualização 3D e a inclinação da câmera são melhorias importantes que foram feitas na API V2.



Figura 22.7 – Parâmetro tilt (inclinação) com os valores 0°, 45° e 90°.



Figura 22.8 – Mapa em 3D com o tilt (inclinação) com os valores 0°, 45° e 90°.

22.15 Monitorando os eventos do mapa

Todas as vezes que a posição do mapa é alterada, podemos monitorar esse evento e recuperar em tempo de execução todas as propriedades da nova posição, como a latitude e longitude, zoom, tilt e bearing. Para isso, basta chamar o método `GoogleMap.setOnCameraChangeListener(OnCameraChangeListener)` e implementar o método `onCameraChange(position)` da interface `OnCameraChangeListener`.

```

@Override
public void onCameraChange(CameraPosition position) {
    // Mudou algo na CameraPosition
}

```

Outro método utilizado para monitorar os eventos é o `GoogleMap.setOnMapClickListener(OnMapClickListener)`. Nesse caso, ao tocar no mapa, o método `onMapClick(LatLng)` da interface `OnMapClickListener` será chamado, o que permite recuperar a localização exata do toque.

```

@Override
public void onMapClick(LatLng latLng) {
    Toast.makeText(getContext(), "Click: " + latLng, Toast.LENGTH_SHORT).show();
    CameraUpdate update = CameraUpdateFactory.newLatLng(latLng);
    map.animateCamera(update);
}

```

22.16 Marcadores

Os marcadores permitem adicionar figuras ao mapa, para marcar os locais de interesse, a fim de exibir informações importantes e interagir com o usuário. Um marcador é representado pela classe `Marker` e criado com o método `GoogleMap.addMarker(markerOptions)`, conforme demonstrado neste código:

```

// Adiciona um marcador
private void adicionarMarcador(GoogleMap map, LatLng latLng) {
    MarkerOptions markerOptions = new MarkerOptions();
    markerOptions.position(latLng).title("Meu Marcador").snippet("Livro Android");
    Marker marker = map.addMarker(markerOptions);
}

```

Para adicionar um marcador, utilizamos um objeto do tipo `MarkerOptions`, e para configurar a localização do marcador, o método `position(LatLng)`. O retorno do método `GoogleMap.addMarker(markerOptions)` é o objeto `Marker`, que vai conter as propriedades que lhe foram atribuídas. Por padrão, o marcador tem o ícone do Google Maps, mas caso seja necessário alterar o ícone chame o método `MarkerOption.setIcon(BitmapDescriptor)`:

```

// Adiciona um marcador
private void adicionarMarcador(GoogleMap map, LatLng latLng) {
    MarkerOptions markerOptions = new MarkerOptions();
    markerOptions.position(latLng).title("Meu Marcador").snippet("Livro Android");
    markerOptions.icon(BitmapDescriptorFactory.fromResource(R.mipmap.ic_launcher));
    Marker marker = map.addMarker(markerOptions);
}

```


Ao clicar no marcador, uma janela é exibida com os textos definidos nas propriedades `title` e `snippet`. A figura 22.9 mostra essa janela, que é chamada de **info window**. Para customizar a janela **info window**, utilize o método `GoogleMap.setInfoWindowAdapter(adapter)` passando como parâmetro um objeto que implemente a interface `InfoWindowAdapter`. Essa interface contém dois métodos: `getInfoWindow()` e `getInfoContents()`. O método `getInfoWindow()` deve retornar a view que é a borda da janela, além de todo o conteúdo interno. Mas caso você deseje alterar apenas o conteúdo interno da janela, como título e descrição, utilize o método `getInfoContents()`.



Figura 22.9 – Janela-padrão ao clicar em um marcador.

Este trecho de código demonstra como implementar o método `getInfoContents()`, que cria uma view dinamicamente:

```
// Adiciona um marcador
private void adicionarMarcador(GoogleMap map, LatLng latLng) {
    MarkerOptions markerOptions = new MarkerOptions();
    markerOptions.position(latLng).title("Meu Marcador").snippet("Livro Android");
    markerOptions.icon(BitmapDescriptorFactory.fromResource(R.drawable.ic_launcher));
    Marker marker = map.addMarker(markerOptions);
    // Customiza a janela ao clicar em um marcador
    map.setInfoWindowAdapter(new InfoWindowAdapter() {
        @Override
        public View getInfoWindow(Marker marker) {
            // Janela completa com conteúdo e borda (retorna null para deixar a janela-padrão)
            return null;
        }
    })
}
```

```

@Override
public View getInfoContents(Marker marker) {
    // View com o conteúdo
    LinearLayout linear = new LinearLayout(getContext());
    linear.setLayoutParams(new LayoutParams(LayoutParams.WRAP_CONTENT,
        LayoutParams.WRAP_CONTENT));
    linear.setOrientation(LinearLayout.VERTICAL);
    TextView t = new TextView(getContext());
    t.setText("*** View customizada *** ");
    t.setTextColor(Color.BLACK);
    t.setGravity(Gravity.CENTER);
    linear.addView(t);
    TextView tTitle = new TextView(getContext());
    tTitle.setText(marker.getTitle());
    tTitle.setTextColor(Color.RED);
    linear.addView(tTitle);
    TextView tSnippet = new TextView(getContext());
    tSnippet.setText(marker.getSnippet());
    tSnippet.setTextColor(Color.BLUE);
    linear.addView(tSnippet);
    return linear;
}
});
}

```

A figura 22.10 demonstra o resultado desse código, cuja janela interna, que exibe os textos title e snippet, está customizada. Repare que, no código, o método `getInfoWindow(marker)` está retornando null, porque não queremos alterar a borda da janela. A borda da janela deve ser uma imagem com a extensão `.9` (NinePatch ou 9-patch), que é um tipo de imagem especial que pode esticar sem perder a resolução.

Agora, vamos alterar o método `getInfoWindow(marker)` para criar a view completa, mas delegaremos o trabalho para o método `getInfoContents(marker)` que acabamos de implementar. Depois de criar a view, uma imagem de fundo arredondada é configurada para ser a borda da janela, conforme demonstrado a seguir:

```

public View getInfoWindow(Marker marker) {
    LinearLayout linear = (LinearLayout) this.getInfoContents(marker); // Cria a view
    // Borda imagem 9-patch
    linear.setBackgroundResource(R.drawable.janela_marker); // Imagem de fundo 9-patch
    return linear;
}

```



Figura 22.10 – Janela com o conteúdo customizado.

Ao executar o exemplo, podemos visualizar a borda customizada da janela, conforme mostra a figura 22.11. Na figura da esquerda, você pode visualizar em detalhes a imagem 9-patch que foi utilizada, e na direita, o resultado.



Figura 22.11 – Janela com a borda customizada.

Nota: o método `getInfoContents()` somente é chamado se o método `getInfoWindow()` retornar `null`. Se o método `getInfoWindow()` retornar uma `view`, deverá retornar uma `view` completa, com a borda da janela e o conteúdo. Se você não retornar a borda, que é a imagem de fundo, os textos voarão sem nenhum contorno ao redor.

Já aprendemos a customizar a aparência da janela de detalhes do marcador, que é chamada de **info window**. Agora vamos aprender a monitorar o clique nesse marcador. Para isso, basta utilizar o método `setOnMarkerClickListener(listener)`:

```
// Evento ao clicar no marcador
map.setOnMarkerClickListener(new OnMarkerClickListener() {
    @Override
    public boolean onMarkerClick(Marker marker) {
        LatLng lartLng = marker.getPosition();
        Toast.makeText(getContext(), "Marcador: " + marker.getTitle() + " > " + lartLng,
            Toast.LENGTH_SHORT).show();
        return false;
    }
});
```

Porém, o método `onMarkerClick(marker)` é chamado quando o usuário clica em um marcador, mas quando a janela **info window** ainda está fechada. Para tratar o evento de clique na janela que está aberta, utilize o método `setOnInfoWindowClickListener(listener)`.

```
// Evento ao clicar no marcador
map.setOnInfoWindowClickListener(new OnInfoWindowClickListener() {
    @Override
    public void onInfoWindowClick(Marker marker) {
        LatLng lartLng = marker.getPosition();
        Toast.makeText(getContext(), "Clicou no: " + marker.getTitle() + " > "
            + lartLng, Toast.LENGTH_SHORT).show();
    }
});
```

Nota: a janela `info window` é renderizada como uma imagem, portanto você não pode adicionar botões a essa `view`, pois não teria efeito algum. Para tratar o evento de clique nessa janela, utilize o método `setOnMarkerClickListener(listener)`.

22.17 Polyline – desenhando uma linha no mapa

A classe `Polyline` permite desenhar uma linha no mapa, que, na verdade, é uma sequência de coordenadas.

No exemplo anterior, criamos o método `adicionarMarcador(GoogleMap, LatLng)`, o qual adiciona um marcador à localização desejada. Podemos chamar esse método quantas vezes forem necessárias para adicionar vários marcadores. Por exemplo, o seguinte trecho de código adiciona dois marcadores:

```
LatLng latLng = new LatLng(-23.564224, -46.653156);
LatLng latLng2 = new LatLng(-23.555696, -46.662627);
// Adiciona os marcadores
adicionarMarcador(map, latLng);
adicionarMarcador(map, latLng2);
```

Nesse exemplo, as duas coordenadas são da Av. Paulista – SP, mas em locais diferentes. Agora que temos essas duas localizações, podemos brincar com a classe `Polyline` e desenhar uma linha entre esses dois pontos. O código-fonte a seguir mostra como desenhar essa linha entre dois pontos conhecidos.

```
private void adicionaPolyline(GoogleMap map2, LatLng latLng, LatLng latLng2) {
    // Desenha uma linha entre dois pontos
    PolylineOptions line = new PolylineOptions();
    line.add(new LatLng(latLng.latitude, latLng.longitude));
    line.add(new LatLng(latLng2.latitude, latLng2.longitude));
    line.color(Color.BLUE);
    Polyline polyline = map.addPolyline(line);
    polyline.setGeodesic(true);
}
```

A figura 22.12 exibe o resultado desse trecho de código com uma linha entre os dois marcadores. Apenas para brincar novamente com o conceito de inclinação da câmera do mapa, a figura da esquerda está com a propriedade `tilt = 0` e da direita com o `tilt = 90`. Note que, embora o zoom da figura da direita seja maior (mais próximo), o parâmetro `tilt` permite inclinar a câmera, aumentando o horizonte de visualização.

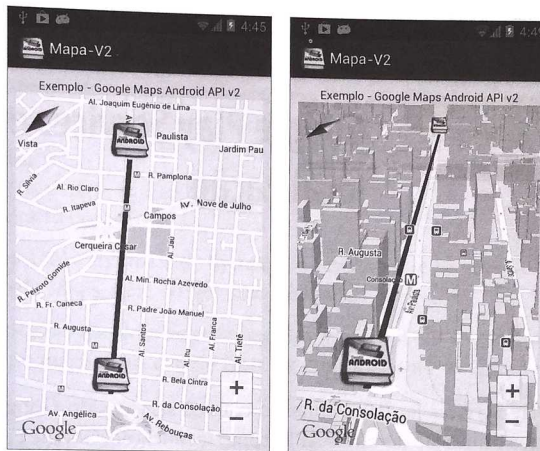


Figura 22.12 – Dois marcadores e uma linha.

22.18 Links úteis

Neste capítulo, estudamos a API de mapas V2. O assunto é extenso e interessante, portanto recomendo que você continue os seus estudos. Um bom começo é a documentação oficial.

- **Google Services – Google Maps Android API v2**
<https://developer.android.com/google/play-services/maps.html>
- **Android Docs – Google Maps Android API v2**
<https://developers.google.com/maps/documentation/android/>
- **Android Docs – Displaying The Debug Certificate Fingerprint**
<https://developers.google.com/maps/documentation/android/start>



CAPÍTULO 23

Google Play Services e localização

O Google Play Services facilita a conexão com os serviços do Google e, consequentemente, a utilização de várias APIs, como localização, Google Fit, Google+, Google Drive, Games etc.

Neste capítulo, vamos aprender a nos conectar aos serviços do Google pelo Play Services e a utilizar a API de localização, conhecida como Fused Location Provider. Uma vez que você aprender a se conectar ao Google Play Services, verá que várias outras APIs podem ser utilizadas da mesma forma. No capítulo 35, sobre sensores, até vamos brincar com o Google Fit.

23.1 Introdução

Já sabemos que o Google Play Services é um aplicativo distribuído pelo Google Play que apresenta funcionalidades essenciais para se comunicar com os serviços do Google, além de várias APIs.

Até o momento, vimos que a API de mapas V2 faz parte do Google Play Services, porém na prática não foi necessário se conectar aos servidores do Google. Neste capítulo, faremos essa conexão para obter a localização por GPS. Lembrando que naturalmente o Google Play Services precisa estar declarado como dependência do projeto.



app/build.gradle

```
compile 'com.google.android.gms:play-services:7.0.0'
```

23.2 My Location

Uma funcionalidade muito comum do aplicativo do Google Maps é a bolinha azul que representa a localização do usuário. Pela API, essa bolinha azul pode ser ativada com apenas uma linha de código:

```
GoogleMap map = ...;
// Exibe a localização do usuário
map.setMyLocationEnabled(true);
```

Essa bolinha azul é chamada de **My Location**, e acredito que, se você utiliza o aplicativo do Google Maps, sabe do que estou falando.

Porém, se você deseja obter as coordenadas, e não apenas mostrar a localização do usuário no mapa, é necessário ligar o monitoramento de GPS.

23.3 Monitorando o GPS (à moda antiga)

Antigamente, antes de existir o Google Play Services, a classe `LocationManager` era utilizada para obter a localização do GPS. Explicarei rapidamente como funcionava o monitoramento de GPS com essa classe, para focarmos logo no Google Play Services.

Tudo era feito com a classe `LocationManager` e o método `requestLocationUpdates(...)`.

```
LocationManager locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0, implementação
de LocationListener aqui...);
```

O monitoramento do GPS costuma ser ligado e desligado nos métodos `onResume()` e `onPause()` da `activity`, para vincular o funcionamento do GPS com o ciclo de vida da `activity`. É muito importante cancelar o listener do GPS quando a `activity` entra em estado de **pause** ou **stop**, para não gastar muita bateria sem necessidade. Lembre-se de que o GPS é um dos principais vilões da bateria, portanto o utilize somente quando necessário:

```
@Override
protected void onResume() {
    super.onResume(); // Liga o GPS
    locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0, this);
}
@Override
protected void onPause() {
    super.onPause(); // Desliga o GPS
    locationManager.removeUpdates(this);
}
```


A seguir, veja a explicação de cada parâmetro do método `requestLocationUpdates(...)`:

Parâmetro	Descrição
String provedor	Nome do provedor de localização (location provider). Podemos utilizar as constantes <code>LocationManager.GPS_PROVIDER</code> ou <code>LocationManager.NETWORK_PROVIDER</code> .
long tempoMínimo	Tempo em milissegundos que representa o intervalo mínimo de tempo em que a aplicação deve receber atualizações sobre as localizações. Se o valor for 0 (zero), o método será chamado sempre que a localização for alterada.
float distanciaMínima	Distância em metros que representa a distância mínima necessária a ser percorrida para a aplicação receber atualizações sobre as localizações. Se o valor for 0 (zero), o método será chamado sempre que a localização for alterada.
LocationListener listener	Implementação da interface <code>LocationListener</code> que receberá as atualizações das coordenadas pelo método <code>onLocationUpdate(location)</code> .

Feito isso, sempre que o Android tiver uma nova localização, ele chamará o método `onLocationChanged(location)`, passando como parâmetro um objeto `android.location.Location`.

```
@Override
public void onLocationChanged(Location location) {
    // Faça algo com esta coordenada aqui
    double latitude = location.getLatitude();
    double longitude = location.getLongitude();
}
```

Basicamente é isso. Como vimos, é simples. O mais importante que você precisa saber de tudo isso é o conceito de provedor de GPS (GPS Provider). O método `requestLocationUpdates(...)` recebe como parâmetro o provedor de GPS, que pode ser uma das duas constantes `LocationManager.GPS_PROVIDER` ou `LocationManager.NETWORK_PROVIDER`.

O provedor `LocationManager.GPS_PROVIDER` utiliza o hardware do GPS, ou seja, tem alta precisão. Porém, esse provedor geralmente demora em retornar o primeiro resultado, tempo que pode variar desde 30 segundos a alguns minutos. Por isso, nem sempre esse provedor é o mais indicado.

O provedor `LocationManager.NETWORK_PROVIDER` utiliza a triangulação de antenas da operadora para descobrir a localização do usuário e também o sinal da rede Wi-Fi. A grande vantagem desse provedor é que ele retorna a primeira localização rapidamente. No entanto, pode retornar a localização com metros ou até quilômetros de diferença, pois a localização por meio de triangulação de antenas calcula um valor aproximado.

Mas, então, qual provedor utilizar? O provedor `GPS_PROVIDER` é preciso, porém é lento; já o provedor `NETWORK_PROVIDER` não é preciso, porém é rápido. A resposta, no caso de utilizar a API antiga, era: Utilize os dois!

```
// Liga o GPS
LocationManager locationManager = . . . ;
locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0, this);
locationManager.requestLocationUpdates(LocationManager.NETWORK_PROVIDER, 0, 0, this);
```

Dessa forma, sabemos que o provedor `LocationManager.NETWORK_PROVIDER` vai retornar a localização de forma mais rápida, o que vai permitir que a localização no mapa seja atualizada rapidamente. Não importa que a localização não seja tão precisa, pois é melhor mostrar algo para o usuário do que não mostrar nada. Passado um tempo, o provedor `LocationManager.GPS_PROVIDER` que utiliza o hardware pode retornar uma localização mais precisa, que é utilizada para atualizar o mapa. Nesse momento, sabemos que o GPS por hardware já está funcionando, portanto podemos cancelar o monitoramento do GPS por triangulação de antenas para economizar recursos. Agora que já sabemos o que é um provedor de GPS, vamos então estudar o provedor Fused Location Provider.

23.4 Monitorando o GPS (Fused Location Provider)

Conforme acabamos de estudar, existem dois provedores de GPS no Android: `GPS_PROVIDER` e `NETWORK_PROVIDER`. Embora pareça simples utilizar o GPS, na prática não é. A maioria dos aplicativos que precisavam utilizar o GPS não fazia as otimizações necessárias e acabava utilizando recursos demais e gastando muita bateria. Na verdade, utilizar o GPS de forma otimizada era um grande desafio.

Foi então que nasceu o Fused Location Provider, um provedor de GPS criado pelo Google, que faz parte do Google Play Services. Esse provedor encapsula o acesso aos provedores `GPS_PROVIDER` e `NETWORK_PROVIDER`, aplicando todas as otimizações e boas práticas necessárias. Resumindo, o Google foi lá e resolveu o problema. Com a nova API de localização do Google Play Services, é possível utilizar o GPS de forma simples e transparente, com a garantia de otimizar ao máximo o uso dos recursos e bateria.

Contudo, antes de utilizar a nova API, é necessário se conectar ao Google Play Services; sendo assim, vamos aprender a fazer isso.



23.5 Conectando-se ao Google Play Services

Para utilizar qualquer uma das APIs gerenciadas pelo Google Play Services, é necessário se conectar aos serviços do Google. A classe responsável por fazer essa conexão é a `GoogleApiClient`, que encapsula toda a comunicação com os servidores do Google. Isso é feito com o seguinte código:

```
GoogleApiClient mGoogleApiClient = new GoogleApiClient.Builder(this)
    .addConnectionCallbacks(this)
    .addOnConnectionFailedListener(this)
    .addApi(LocationServices.API)
    .build();
```

Veja que foi informada a API de localização com o método `addApi(LocationServices.API)`. Porém, como dissemos antes, existem várias APIs que fazem parte do Google Play Services. Portanto, apenas para exemplificar, o seguinte trecho de código mostra como se conectar ao serviço do Google Drive. Basta adicionar a linha `addApi(Drive.API)`.

```
GoogleApiClient mGoogleApiClient = new GoogleApiClient.Builder(this)
    .addConnectionCallbacks(this)
    .addOnConnectionFailedListener(this)
    .addApi(Drive.API)
    .addScope(Drive.SCOPE_FILE)
    .build();
```

Depois de construir o objeto `GoogleApiClient`, o que geralmente é feito no método `onCreate()` da `activity` ou `fragment`, podemos chamar o método `GoogleApiClient.connect()` para realizar a conexão. Segundo a documentação oficial, o Google recomenda realizar a conexão no método `onStart()` da `activity` e fechar a conexão no método `onStop()`, conforme demonstrado neste trecho de código:

```
@Override
protected void onStart() {
    super.onStart();
    mGoogleApiClient.connect();
}

@Override
protected void onStop() {
    super.onStop();
    mGoogleApiClient.disconnect();
}
```

Para receber o resultado da conexão, seja sucesso ou falha, devemos implementar os métodos das interfaces `ConnectionCallbacks` e `OnConnectionFailedListener`.

```

@Override
public void onConnected(Bundle connectionHint) {
    // Conectado ao Google Play Services!
    // Podemos utilizar qualquer API agora.
}

@Override
public void onConnectionSuspended(int cause) {
    // A conexão foi interrompida.
    // A aplicação precisa aguardar até a conexão ser restabelecida
}

@Override
public void onConnectionFailed(ConnectionResult result) {
    // Erro na conexão.
    // Pode ser uma configuração inválida ou falta de conectividade no dispositivo.
}

```

Agora que sabemos como nos conectar aos serviços do Google, vamos estudar a API de localização do Google Play Services.

23.6 Obtendo a última localização de forma eficiente

Com o Google Play Services, uma das principais melhorias feitas na API de localização é a facilidade de se obter a última localização do usuário de forma rápida. Uma vez que a aplicação está conectada aos serviços do Google, basta utilizar uma linha de código para recuperar a última localização conhecida pelo sistema.

```

GoogleApiClient mGoogleApiClient = . . .
Location location = LocationServices.FusedLocationApi.getLastLocation(mGoogleApiClient);

```

Para praticar o conceito, vamos criar o projeto **MyLocation**. No entanto, se preferir, abra o projeto pronto no Android Studio. No layout da activity vamos inserir o mapa com a classe `SupportMapFragment`.

```

 /res/layout/activity_main.xml

<LinearLayout android:orientation="vertical" . . .>
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Mostra como obter a última localização pela API."/>
    <fragment android:id="@+id/map"
        android:layout_width="match_parent" android:layout_height="match_parent"
        class="com.google.android.gms.maps.SupportMapFragment" />
</LinearLayout>

```

Quando a aplicação abrir, o mapa será mostrado à localização padrão. Vamos inserir um botão na action bar que, ao ser clicado, vai recuperar a última localização conhecida e centralizar o mapa nesse local. Esse botão vai simular o método `setMyLocationEnabled(boolean)` da classe `GoogleMap`.

```

<?xml version="1.0" encoding="utf-8" android:label="@string/app_name"
    />
<menu
    android:res_id="@id/action_my_location"
    android:showAsAction="always"
    />

```

Nota: para este projeto funcionar, é preciso declarar todas as permissões e configurar o arquivo de manifesto, como fizemos no capítulo anterior. Atente-se para a chave `API Key`, que é gerada na página de console do Google. No meu caso, o pacote do projeto é `br.com.livroandroid.location`, que é diferente do pacote `br.com.livroandroid.carros` do aplicativo dos carros. Portanto, é necessário gerar outra `API Key`. No dia a dia, cada aplicativo tem um pacote diferente, ou seja, cada um deve ter a sua `API Key`.

No código da `activity`, vamos obter o `SupportMapFragment`, como já fizemos anteriormente, e também vamos nos conectar ao Google Play Services. Tudo isso já aprendemos a fazer, portanto o código a seguir é uma revisão. Leia atentamente, pois, apesar de extenso, o código está comentado.

 **MainActivity.java**

```

public class MainActivity extends AppCompatActivity implements OnMapReadyCallback,
    GoogleApiClient.ConnectionCallbacks, GoogleApiClient.OnConnectionFailedListener {
    private static final String TAG = "livroandroid";
    protected GoogleMap map;
    private SupportMapFragment mapFragment;
    private GoogleApiClient mGoogleApiClient;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Fragment de mapa
        mapFragment = (SupportMapFragment) getSupportFragmentManager().findFragmentById(R.id.map);
        mapFragment.getMapAsync(this);
    }
}

```

```
// Configura o objeto GoogleApiClient
mGoogleApiClient = new GoogleApiClient.Builder(this)
    .addConnectionCallbacks(this)
    .addOnConnectionFailedListener(this)
    .addApi(LocationServices.API)
    .build();
}

@Override
public void onMapReady(GoogleMap map) {
    Log.d(TAG, "onMapReady: " + map);
    this.map = map;
    // Configura o tipo do mapa
    map.setMapType(GoogleMap.MAP_TYPE_NORMAL);
}

@Override
protected void onStart() {
    super.onStart();
    // Conecta ao Google Play Services
    mGoogleApiClient.connect();
}

@Override
protected void onStop() {
    // Desconecta do Google Play Services
    mGoogleApiClient.disconnect();
    super.onStop();
}

@Override
public void onConnected(Bundle bundle) {
    toast("Conectado ao Google Play Services!");
}

@Override
public void onConnectionSuspended(int cause) {
    toast("Conexão interrompida.");
}

@Override
public void onConnectionFailed(ConnectionResult connectionResult) {
    toast("Erro ao conectar: " + connectionResult);
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return super.onCreateOptionsMenu(menu);
}
```

```

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_my_location:
            // Obtém última localização
            Location l = LocationServices.FusedLocationApi.getLastLocation(
                mContext.getSystemService(Context.LOCATION_SERVICE));
            Log.d(TAG, "lastLocation: " + l);
            // Atualiza a localização do mapa
            setMapLocation(l);
            return true;
        }
    return super.onOptionsItemSelected(item);
}

// Atualiza a coordenada do mapa
private void setMapLocation(Location l) {
    if (map != null && l != null) {
        LatLng latLng = new LatLng(l.getLatitude(), l.getLongitude());
        CameraUpdate update = CameraUpdateFactory.newLatLngZoom(latLng, 15);
        map.animateCamera(update);
        Log.d(TAG, "setMapLocation: " + l);
        TextView text = (TextView) findViewById(R.id.text);
        text.setText(String.format("Lat/Lnt %f/%f, provider: %s", l.getLatitude(),
            l.getLongitude(), l.getProvider()));
        // Desenha uma bolinha vermelha
        CircleOptions circle = new CircleOptions().center(latLng);
        circle.fillColor(Color.RED);
        circle.radius(25); // Em metros
        map.clear();
        map.addCircle(circle);
    }
}

private void toast(String s) {
    Toast.makeText(getApplicationContext(), s, Toast.LENGTH_SHORT).show();
}
}

```

Ao executar o exemplo, o mapa é mostrado na localização padrão, e ao clicar no botão **minha localização** que adicionamos à action bar o mapa será posicionado na última localização conhecida. Como um plus para o exemplo, foi mostrado como desenhar um círculo com a classe `CircleOptions` e o método `addCircle(circle)`. Fiz isso para simular o que o método `setMyLocation(boolean)` faz. A figura 23.1 mostra o resultado deste exemplo.

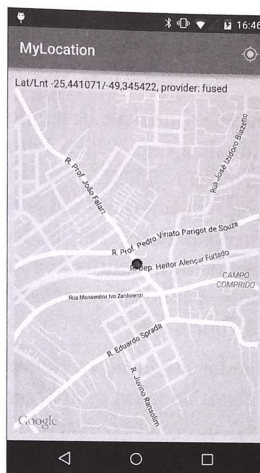


Figura 23.1 – Obtendo a última localização.

Dica: embora o método `setMyLocationEnabled(boolean)` da classe `GoogleMap` faça a mesma coisa que esse exemplo acabou de fazer, aprendemos a obter a localização pela API. O método `setMyLocationEnabled(boolean)` apenas mostra o botão de `my location` e desenha a famosa bolinha azul no mapa, mas não fornece a localização ao aplicativo. Lembre-se de que este projeto também mostrou como se conectar ao Google Play Services.

23.7 API de localização do Google Play Services

Para iniciar o monitoramento de GPS, é necessário criar um objeto `LocationRequest`, que contém as configurações referentes à precisão e ao intervalo de tempo com os quais você deseja receber as coordenadas.

```
LocationRequest mLocationRequest = new LocationRequest();  
mLocationRequest.setInterval(10000); // 10 segundos  
mLocationRequest.setFastestInterval(5000); // 5 segundos  
mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
```

O método `setPriority(int)` define a precisão do GPS. As constantes mais utilizadas são `PRIORITY_HIGH_ACCURACY` e `PRIORITY_LOW_POWER`. A primeira tem como objetivo retornar a localização mais precisa possível. A segunda visa obter uma localização

aproximada a fim de economizar bateria. Outra constante interessante é a `PRIORITY_NO_POWER`, que não inicia o monitoramento do GPS, mas garante que, se outra aplicação no sistema operacional o fizer, você receberá essa localização.

O método `setInterval(long)` recebe o intervalo de tempo em milissegundos em que a aplicação deseja receber atualizações do GPS. Porém, o Google Play Services compartilha os recursos com todas as aplicações instaladas, e, se alguma aplicação solicitou um intervalo de tempo menor que a sua, o Android vai enviar os dados para sua aplicação nesse intervalo menor. Por um lado, o Google Play Services otimiza os recursos e entrega de forma eficiente a nova localização para todas as aplicações; por outro lado, sua aplicação pode receber as atualizações em um intervalo de tempo menor do que o esperado.

Como o Android pode mandar atualizações freneticamente, dependendo da configuração do `setInterval(long)` de todas as aplicações instaladas, foi criado o método `setFastestInterval(long)`, que recebe o intervalo mínimo no qual sua aplicação consegue receber e tratar os eventos corretamente, sem afetar a performance da aplicação. Dessa forma, é garantido que você não receba nenhuma atualização antes desse intervalo.

Depois de criar o objeto `LocationRequest` com as devidas configurações, podemos utilizar os seguintes métodos para iniciar ou parar o GPS.

```
GoogleApiClient mGoogleApiClient = . . .
protected void startLocationUpdates() {
    LocationServices.FusedLocationApi.requestLocationUpdates(
        mGoogleApiClient, mLocationRequest, this);
}
protected void stopLocationUpdates() {
    LocationServices.FusedLocationApi.removeLocationUpdates(
        mGoogleApiClient, this);
}
```

O momento correto de iniciar o GPS é logo depois de a conexão com o Google Play Services ser realizada, portanto isso deve ser feito no método `onConnected()`.

```
@Override
public void onConnected(Bundle connectionHint) {
    // Conectado ao Google Play Services!
    startLocationUpdates(); // Inicia o GPS
}
```

Se quisermos parar o GPS podemos utilizar os métodos do ciclo de vida da `activity` ou `fragment`, como o `onPause()` ou `onStop()`.

```

@Override
protected void onPause() {
    super.onPause();
    stopLocationUpdates(); // Para o GPS
}

```

Por último, para receber as localizações, devemos implementar o método `onLocationChanged(location)` da interface `LocationListener`. Pronto, isso é tudo!

```

@Override
public void onLocationChanged(Location location) {
    double latitude = location.getLatitude();
    double longitude = location.getLongitude();
    // Faça algo com a latitude/longitude aqui.
}

```

Para ver um exemplo completo de código que utiliza o serviço de localização do Google Play Services, abra o projeto **LocationUpdates** no Android Studio. Basicamente, esse projeto é idêntico ao exemplo que fizemos anteriormente, mas eu removi o botão da action bar que mostrava a última localização.

O objetivo deste exemplo é mostrar o mapa, conectar-se ao Google Play Services e ligar o GPS. Sempre que uma nova localização for recebida, a posição do mapa será atualizada, assim como a bolinha (circle) que desenhamos no mapa para indicar a atual posição. O `TextView` que está no layout vai mostrar a latitude e longitude, assim como a última hora em que o aplicativo recebeu uma localização.



MainActivity.java

```

public class MainActivity implements OnMapReadyCallback, GoogleApiClient.
    ConnectionCallbacks, GoogleApiClient.OnConnectionFailedListener,
    com.google.android.gms.location.LocationListener {
    . . .
    @Override
    protected void onCreate(Bundle savedInstanceState) { . . . }
    public void onMapReady(GoogleMap map) { . . . }
    protected void onStart() { . . . }
    }
    @Override
    protected void onStop() {
        // Para o GPS
        stopLocationUpdates();
        // Desconecta do Google Play Services
        mGoogleApiClient.disconnect();
        super.onStop();
    }
}

```

```

    }
    @Override
    public void onConnected(Bundle bundle) {
        toast("Conectado ao Google Play Services!");
        // Inicia o GPS
        startLocationUpdates();
    }
    public void onConnectionSuspended(int cause) { . . . }
    public void onConnectionFailed(ConnectionResult connectionResult) { . . . }
    // Atualiza a coordenada do mapa
    private void setMapLocation(Location l) {
        . . . // tudo igual antes ...
    }
    . . .
    /** TUDO IGUAL AO EXEMPLO ANTERIOR "My Location"
     * O que muda é essa parte daqui para baixo...
     * Adicionados métodos para fazer start e stop do GPS
     * Veja também os métodos onConnected(bundle) e onStop() */
    protected void startLocationUpdates() {
        Log.d(TAG, "startLocationUpdates()");
        LocationRequest mLocationRequest = new LocationRequest();
        mLocationRequest.setInterval(10000);
        mLocationRequest.setFastestInterval(5000);
        mLocationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
        LocationServices.FusedLocationApi.requestLocationUpdates(mGoogleApiClient,
            mLocationRequest, this);
    }
    protected void stopLocationUpdates() {
        Log.d(TAG, "stopLocationUpdates()");
        LocationServices.FusedLocationApi.removeLocationUpdates(mGoogleApiClient, this);
    }
    @Override
    public void onLocationChanged(Location location) {
        Log.d(TAG, "onLocationChanged(): " + location);
        // Nova localização: atualiza a interface
        setMapLocation(location);
    }
}

```

Recomendo executar esse exemplo em um dispositivo real e fazer seus testes. Esse código atualiza a posição do mapa sempre que uma nova localização é recebida do GPS. Note que a activity implementa a interface `LocationListener` e o método `onLocationChanged()` para receber as localizações.

Nota: lembre-se de que para utilizar o GPS é necessário declarar as permissões `ACCESS_FINE_LOCATION` (GPS por hardware) ou `ACCESS_COARSE_LOCATION` (Wi-Fi ou triangulação de antenas), dependendo das configurações de precisão feitas no objeto `LocationRequest`.

23.8 Desenhando uma rota entre dois pontos

Um requisito muito comum em aplicativos é a necessidade de traçar uma rota entre dois lugares conhecidos. Isso pode ser feito de duas maneiras: a primeira é utilizando uma intent para pedir ao aplicativo do Google Maps que desenhe a rota, a segunda é desenhar a rota manualmente por programação.

Para a grande maioria das aplicações, não acredito que seja viável traçar uma rota desenhando manualmente sobre o mapa. A justificativa é simples: você provavelmente não fará melhor do que o aplicativo do Google Maps faz. Então, a não ser que você esteja fazendo um aplicativo de mapas customizado, a maneira mais fácil e recomendada de traçar uma rota é disparar uma intent, informando a coordenada inicial e a final. O aplicativo nativo do Google Maps vai interceptar a intent e se encarregará do trabalho de traçar a rota.

O trecho de código a seguir mostra como disparar uma intent para mostrar uma rota entre duas coordenadas:

```
// Dispara uma intent para traçar uma rota
String origem = "-25.443195, -49.280977";
String destino = "-25.442207, -49.278403";
String url = "http://maps.google.com/maps?f=d&saddr="+origem+"&daddr="+destino+"&hl=pt";
startActivity(new Intent(Intent.ACTION_VIEW, Uri.parse(url)));
```

A figura 23.2 exibe o resultado:

No entanto, caso seja realmente necessário traçar a rota manualmente, você poderá fazer uma requisição HTTP neste web service do Google, informando as coordenadas de origem e destino:

```
// Descobrir a rota
String origem = "-25.443195, -49.280977";
String destino = "-25.442207, -49.278403";
String url = "http://maps.googleapis.com/maps/api/directions/json?origin=" + origem +
    "&destination="+destino+"&sensor=true&mode=driving";

String json = HttpHelper.doGet(url);
```



Figura 23.2 – Desenhando uma rota.

O retorno será um JSON que contém as coordenadas da rota, separadas por pontos com a latitude e a longitude. Depois de fazer o parser e obter a lista de coordenadas, você poderá fazer um loop e desenhar várias linhas utilizando o conceito de *PolyLine*, que estudamos no capítulo anterior. Se precisar de ajuda, faça a busca no Google por **Google Directions API**. Com certeza, encontrará posts em blogs com bons exemplos de código.

Agora vamos voltar a falar um pouco do projeto dos carros. Na tela do mapa, lembre-se de que deixamos um botão para traçar a rota, cujo identificador da ação é `R.id.action_location_directions`.

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    ...
    if (item.getItemId() == R.id.action_location_directions) {
        // Posiciona mapa no usuário
        toast("Mostrar rota/direções até a fábrica.");
    }
}
```

O objetivo é desenhar uma rota entre a localização atual e a localização da fábrica do carro. Mas isso vou deixar como exercício para você, caso queira praticar, pois tudo o que você precisa saber para implementar esse tipo de funcionalidade já foi explicado.

23.9 Buscando um endereço

Outra dica interessante que vale a pena mencionar é como converter um endereço para latitude e longitude.

Isso pode ser feito facilmente com a classe `android.location.Geocoder` e o método `getFromLocationName(string)`, que retorna uma lista de objetos do tipo `android.location.Address`, sendo que a classe `Address` contém a latitude e a longitude do local encontrado. O retorno é uma lista de possíveis resultados, assim, geralmente, o primeiro objeto é o mais próximo do resultado:

```
// Para descobrir a latitude e a longitude do endereço
String endereco = "Av Paulista - SP";
Geocoder gc = new Geocoder(this, new Locale("pt", "BR"));
List<Address> list = gc.getFromLocationName(endereco, 10);
Toast.makeText(this, "Retorno: " + list, Toast.LENGTH_SHORT).show();
```

23.10 Links úteis

Neste capítulo, aprendemos a nos conectar ao Google Play Services e utilizar a API de localização. Para continuar seus estudos, separei alguns links interessantes.

- **Google Services – Accessing Google APIs**

<https://developer.android.com/google/auth/api-client.html>

- **Android Training – Making Your App Location Aware**

<https://developer.android.com/training/location/index.html>

- **Google Developers – Google Directions API**

<https://developers.google.com/maps/documentation/directions/>



CAPÍTULO 24

BroadcastReceiver

Este capítulo fala sobre a classe `BroadcastReceiver`, uma das mais importantes na arquitetura do Android. Ela é utilizada para que aplicações possam interceptar eventos de sistema, ou seja, mensagens geradas por uma intent.

Na prática, o broadcast receiver é uma classe que consegue interceptar uma intent, sendo a intent uma mensagem que pode ser enviada pela sua própria aplicação, por uma aplicação de terceiros ou pelo próprio sistema operacional.

Um broadcast receiver sempre executa em segundo plano (background) e não utiliza interface gráfica. Seu objetivo é interceptar uma mensagem (intent) e processá-la sem que o usuário perceba.

24.1 Introdução

A classe `android.content.BroadcastReceiver` é utilizada para interceptar mensagens enviadas por uma intent. Quando a mensagem é interceptada, o método `onReceive(context,intent)` é chamado, e executa em segundo plano, sem mostrar nenhuma tela ao usuário.

Um broadcast receiver pode ser utilizado para enviar e receber mensagens dentro da própria aplicação ou para interceptar eventos de sistema. Um exemplo de evento de sistema é a mensagem `SMS_RECEIVED`, a qual é enviada pelo Android sempre que uma nova mensagem SMS é recebida pelo sistema operacional. Portanto, caso seja interessante para sua aplicação, é possível interceptar essas mensagens, ou seja, esses eventos, a fim de executar alguma aplicação.

Um broadcast receiver consegue interceptar uma intent/mensagem mesmo se a aplicação estiver parada, e esse é um de seus grandes benefícios.

24.2 Configurando um receiver de forma estática

Um receiver pode ser configurado de forma estática no arquivo *AndroidManifest.xml* ou de forma dinâmica no código. Para começar a brincadeira, vamos aprender a configurá-lo de forma estática. A principal vantagem de um receiver ser configurado de forma estática é que ele consegue interceptar o evento mesmo se a aplicação estiver fechada, ou seja, ele consegue acordar a aplicação se necessário.

Para estudarmos o conceito, crie um projeto chamado **HelloReceiver**. No arquivo de layout da activity, vamos apenas colocar um simples botão que vai enviar uma intent.

 /res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="16dp" android:orientation="vertical">
    <Button android:id="@+id/btEnviar" android:text="Enviar"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
</LinearLayout>
```

No código da activity, vamos disparar uma intent por broadcast. Isso é feito com o método `sendBroadcast(intent)`.

 MainActivity.java

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        findViewById(R.id.btEnviar).setOnClickListener(this);
    }
    @Override
    public void onClick(View v) {
        // Dispara a mensagem de broadcast para todas as aplicações instaladas
        sendBroadcast(new Intent("BINGO"));
        Toast.makeText(this, "Intent enviada!", Toast.LENGTH_SHORT).show();
    }
}
```

Veja que este código disparou a seguinte mensagem de broadcast:

```
sendBroadcast(new Intent("BINGO"));
```

Observe que novamente utilizamos uma intent, mas, em vez de utilizar o método `startActivity(intent)`, apenas trocamos por `sendBroadcast(intent)`. A diferença

é que o primeiro tem como objetivo chamar uma *activity* e o segundo tem como objetivo enviar uma mensagem de broadcast para ser interceptada por algum *BroadcastReceiver*. A palavra “broadcast” é um termo comum para uma mensagem que é enviada para todo mundo ao mesmo tempo. Para interceptar essa mensagem de broadcast, vamos criar a classe *HelloReceiver*, que é filha de *android.content.BroadcastReceiver*. Implementar um receiver não requer prática nem habilidade, pois existe somente o método *onReceive(context, intent)*.

HelloReceiver.java

```
...
import android.content.BroadcastReceiver;
public class HelloReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context c, Intent intent) {
        Log.d("livroandroid", " HelloReceiver !!!");
    }
}
```

O parâmetro *intent* do método *onReceive(context, intent)* é exatamente a *intent* que o receiver interceptou, ou seja, é a mensagem que foi enviada por broadcast. E para que tudo isso funcione basta configurar o receiver no arquivo de manifesto. Isso é feito com a tag *<receiver>*, conforme demonstrado a seguir:

AndroidManifest.xml

```
<manifest . . .>
    <application . . .>
        <activity android:name=".MainActivity" . . . />
        <receiver android:name=".HelloReceiver">
            <intent-filter>
                <action android:name="BINGO" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

Observe que a configuração realizada no arquivo *AndroidManifest.xml* é exatamente a mesma para as tags *<activity>* e *<receiver>*, e ambas declaram o nome de uma classe e o *<intent-filter>*, que contém tanto a ação quanto a categoria para a qual a classe deve executar. Dessa forma, quando alguma *intent*/mensagem com a ação BINGO for disparada, a classe *HelloReceiver* será executada em segundo plano sem interferir na tela em que o usuário está trabalhando ou se divertindo.

Pronto, isso é tudo. Para conferir o resultado, execute o projeto no emulador. Ao clicar no botão, a mensagem BINGO será enviada por broadcast. Se tudo correr bem, você verá a seguinte mensagem nos logs do LogCat:

```
D/livroandroid: HelloReceiver !!!
```

Agora que você já foi apresentado a mais uma classe do Android, pense um pouco na arquitetura de Intent e IntentFilter. Lembre-se de que a Intent representa uma mensagem com a intenção de realizar algo, e o IntentFilter é o que filtra essa mensagem por ação e categoria. Agora já sabemos que esse filtro pode ser configurado para executar uma Activity ou um BroadcastReceiver.

Nota: uma vez configurado no arquivo de manifesto, qualquer aplicação pode enviar uma mensagem que será interceptada pelo receiver. Caso queira evitar esse comportamento, configure o receiver com o atributo `android:exported="false"`; assim, o receiver somente irá receber mensagens da sua própria aplicação.

24.3 Configurando um receiver de forma dinâmica

Um BroadcastReceiver pode ser configurado de duas maneiras diferentes.

1. De forma estática, no arquivo *AndroidManifest.xml* com a tag `<receiver>`. Isso é útil quando precisamos interceptar a mensagem mesmo se a aplicação estiver fechada. É o caso de interceptar uma mensagem SMS ou Push, pois geralmente esse tipo de mensagem precisa ser interceptada mesmo com a aplicação fechada.
2. Registrar o receiver dinamicamente na activity. Nesse caso, o receiver fica atrelado ao ciclo de vida da activity, ou seja, o receiver ficará ativo somente enquanto a activity estiver aberta. Esse recurso é muito utilizado para enviar mensagens internas da aplicação. Por exemplo, um receiver pode interceptar um evento de que o estado da conexão com a internet mudou, e enviar uma mensagem local para avisar a activity se a aplicação está online ou offline.

Como registrar um receiver de forma estática foi o exemplo que fizemos no tópico anterior. Agora, vamos criar o projeto **HelloReceiverDinamico** para demonstrar como registrar o receiver dinamicamente. Neste projeto não precisamos configurar nenhuma tag `<receiver>` no arquivo de manifesto, pois faremos tudo programaticamente. A seguir, podemos verificar o código-fonte da activity:



MainActivity.java

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    private BroadcastReceiver helloReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
            Log.d("livroandroid", "HelloReceiver Dinamico!!!");
            // Este receiver é uma classe interna
            // Portanto, ele consegue chamar os métodos da activity.
            TextView text = (TextView) findViewById(R.id.text);
            text.setText("Mensagem recebida pelo HelloReceiver Dinâmico!!!");
        }
    };
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        findViewById(R.id.btEnviar).setOnClickListener(this);
        // Registra o receiver
        registerReceiver(helloReceiver, new IntentFilter("BINGO"));
    }
    @Override
    public void onClick(View v) {
        sendBroadcast(new Intent("BINGO"));
        Toast.makeText(this, "Intent enviada!", Toast.LENGTH_SHORT).show();
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // Cancela o receiver
        unregisterReceiver(helloReceiver);
    }
}
```

No método `onCreate(bundle)` da activity, o receiver está sendo registrado dinamicamente com o método `registerReceiver(receiver, intentFilter)`. O receiver ficará vivo durante todo o ciclo de vida da activity, pois no método `onDestroy()` ele é cancelado com o método `unregisterReceiver(helloReceiver)`. O restante do exemplo é idêntico ao anterior; a diferença é que, ao disparar a mensagem de broadcast, quem vai interceptar é o receiver que foi declarado como uma classe interna dentro da activity.

No arquivo de layout, além do botão para disparar a intent, foi adicionado um `TextView`, utilizado para mostrar uma mensagem que o receiver vai interceptar.



/res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="16dp" android:orientation="vertical">
    <TextView android:id="@+id/text"
        android:text="Clique no botão para disparar um receiver"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <Button android:id="@+id/btEnviar" android:text="Enviar"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
</LinearLayout>
```

A figura 24.1 mostra o resultado deste exemplo. Ao clicar no botão para disparar a mensagem, ela é interceptada pelo receiver dinâmico e o `TextView` é atualizado.

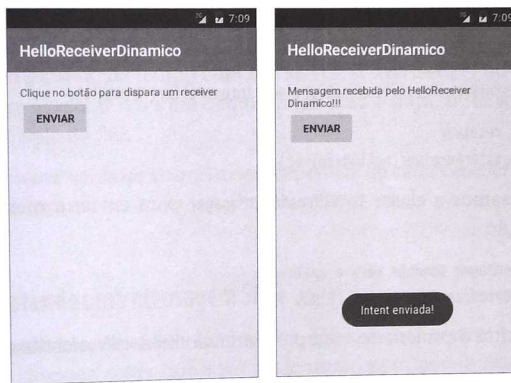


Figura 24.1 – Exemplo de receiver dinâmico.

24.4 Quando utilizar um receiver estático ou dinâmico?

Se você leu com atenção o capítulo, já deve saber a resposta, mas preciso garantir que você entendeu o conceito.

Um receiver estático é configurado no arquivo *AndroidManifest.xml*. A sua principal vantagem é que dessa forma ele vai interceptar os eventos sempre, mesmo se a aplicação estiver fechada.

Já um receiver dinâmico é configurado programaticamente no código da activity. Nesse caso, o seu ciclo de vida está atrelado ao da activity. Isso é geralmente utilizado para enviar mensagens internas dentro da própria aplicação.

24.5 Classe LocalBroadcastManager

Mensagens de broadcast são enviadas para todas as aplicações instaladas, mas às vezes o que queremos é somente enviar uma mensagem de forma local na aplicação. Para isso, é recomendada a classe `android.support.v4.content.LocalBroadcastManager`.

Utilizando a classe `LocalBroadcastManager`, internamente o Android consegue fazer algumas otimizações com relação ao desempenho da troca de mensagens. Outra vantagem é por questões de segurança, pois, se a mensagem for trafegada apenas dentro da aplicação, não corremos risco de nenhuma outra aplicação interceptá-la.

Utilizar a classe `LocalBroadcastManager` é simples, basta obter uma instância da classe:

```
LocalBroadcastManager manager = LocalBroadcastManager.getInstance(this);
```

Feito isso, chamamos os métodos `registerReceiver(receiver,intentFilter)` e `unregisterReceiver(helloReceiver)` da mesma forma que antes:

```
// Registra o receiver de forma local
manager.registerReceiver(helloReceiver, new IntentFilter("BINGO"));

// Cancela o receiver
manager.unregisterReceiver(helloReceiver);
```

Por último, usamos a classe `LocalBroadcastManager` para enviar a mensagem local para a aplicação.

```
// Envia a mensagem somente para a aplicação
manager.sendBroadcast(new Intent("BINGO"));
```

Se quiser, confira o projeto de exemplo chamado `HelloLocalBroadcastManager`.

24.6 Execução de um receiver ao inicializar o sistema operacional

Ao longo de sua vida como programador Android, você vai se deparar algumas vezes com a necessidade de deixar algum serviço executando sempre. Inclusive é preciso iniciar esse serviço quando o celular for ligado.

No Android isso é simples, pois, quando o sistema operacional termina a inicialização, é enviada uma mensagem de broadcast com a ação `android.intent.action.BOOT_COMPLETED`. Nesse caso, é possível interceptar a mensagem e iniciar um `BroadcastReceiver` automaticamente logo depois que o sistema operacional termine a inicialização (boot).

A seguir, podemos verificar a configuração de um receiver que intercepta a ação `BOOT_COMPLETED`.

```
<receiver android:name=".SuaClasseReceiverAqui">
  <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</receiver>
```

Para utilizar a ação `BOOT_COMPLETED`, é necessário declarar a permissão `RECEIVE_BOOT_COMPLETED`.

```
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
```

Para testar essa funcionalidade, instale a aplicação com este receiver configurado e feche o emulador. Depois, inicie o emulador novamente e, quando o boot do sistema terminar, o broadcast receiver vai receber a mensagem.

Esse tipo de recurso é extremamente útil quando precisamos que uma aplicação seja executada sempre. Dessa forma, quando o usuário ligar o celular, podemos executar algum código. Inclusive, podemos manter uma aplicação executando por um longo período de tempo com um `Service` (Capítulo 27) ou agendar uma aplicação para executar em uma determinada data e hora, utilizando o sistema de alarmes (Capítulo 26).

Enfim, o que você vai fazer com isso vai depender de cada caso. O importante é conhecer esse recurso.

24.7 Interceptando uma mensagem SMS

Se você já entendeu o que um receiver faz, provavelmente deve estar tentando adivinhar as diversas ações que você pode interceptar, pois o Android dispara diversas mensagens no formato de intents. Como já foi explicado, a classe `Intent` é o coração do Android, e a classe `BroadcastReceiver` intercepta justamente essas intents. Que dupla, hein?

Apenas para demonstrar mais uma mensagem que podemos interceptar, veja a seguinte configuração do *AndroidManifest.xml*:

```
<receiver android:name=".SuaClasseReceiverAqui ">
  <intent-filter>
    <action android:name="android.provider.Telephony.SMS_RECEIVED" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</receiver>
```

Nesse exemplo, o receiver está configurado para interceptar as mensagens com ação `android.provider.Telephony.SMS_RECEIVED`, as quais são enviadas pelo Android sempre que um novo SMS é recebido. No capítulo 33, sobre SMS, vamos aprender a ler e enviar mensagens por SMS, então vou concluir o exemplo depois. Por ora, mostrei apenas para você entender um caso clássico no qual um receiver é utilizado para interceptar um evento de sistema.

24.8 Ciclo de vida

Um `BroadcastReceiver` é válido somente durante a chamada ao método `onReceive(context, intent)`. Depois disso, o sistema operacional encerrará seu processo para liberar memória. O código do método `onReceive(context, intent)` deve executar brevemente, e assim que o método terminar o Android vai considerar que o `BroadcastReceiver` não está mais ativo e está pronto para ser destruído.

Esse método deve consumir rapidamente a intent (mensagem) recebida e retornar. Caso demore mais de dez segundos para executar, o Android exibirá um erro chamado ANR (Application Not Responding), que nada mais é do que um timeout.

Você provavelmente já deve ter visto esse erro em alguma aplicação, e se isso aconteceu é porque a aplicação fez algo errado. A figura 24.2 mostra o erro ANR, quando um `BroadcastReceiver` demorou mais de dez segundos para executar.

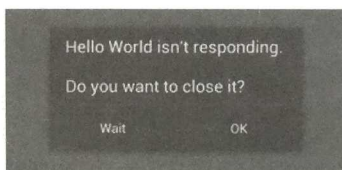


Figura 24.2 – Aplicação não respondendo (ANR).

Nota: um broadcast receiver deve executar em no máximo dez segundos. Caso contrário, o Android vai mostrar o erro ANR. Outra forma de o erro ANR acontecer é quando algum evento gerado dentro de uma activity ou fragment não é tratado em no máximo cinco segundos. Justamente por isso, é importante utilizar threads para tratar os eventos de interfaces dentro da activity.

24.9 Delegando o trabalho para um service

Um `BroadcastReceiver` não deve executar tarefas demoradas durante a chamada ao método `onReceive(context, intent)`. Lembre-se de que todo o código que intercepta o evento deve executar em no máximo dez segundos.

Para exemplificar, digamos que sua aplicação recebeu uma mensagem de Push indicando que existem atualizações no seu servidor. Não importa que tipo de atualização seja essa, mas imagine que seja necessário consultar um web service. Dependendo da velocidade da conexão e da quantidade de dados a serem tráfegados, isso pode demorar um pouco e exceder os dez segundos.

No caso da `activity`, aprendemos que, para não travar a interface do usuário, devemos utilizar `threads`. Porém, no caso do receiver, não podemos utilizar `threads`, pois o sistema operacional vai encerrar o processo do receiver depois de dez segundos, e ao fazer isso ele pode destruir inclusive `threads` que foram criadas, pois o Android não as conhece.

Para solucionar esse problema, foi criada a classe `Service`, que faz parte do ciclo de vida do Android e é especializada em executar serviços em segundo plano por um longo período de tempo. O Android encerrará esse serviço apenas se a memória estiver extremamente baixa e se for necessário encerrar determinados processos para liberar memória. Mesmo se isso acontecer, posteriormente o Android pode inclusive reiniciar o serviço para continuar o que ele estava fazendo. Para mais detalhes, leia o capítulo 27, sobre a classe `Service`.

24.10 Mostrando uma notificação para o usuário

Para finalizar este capítulo, gostaria apenas de lembrar que um `BroadcastReceiver` não deve interagir com o usuário de forma direta, seja exibindo um alerta ou abrindo uma tela sem a sua permissão.

Vou ser mais claro: um receiver não deve atrapalhar o usuário! Nunca chame uma `activity` de dentro de um receiver. Lembre-se de que o usuário pode estar fazendo algo importante, e é bem provável que ele não deseje ser incomodado. Não existe nada mais inconveniente do que aplicações malfeitas que ficam atrapalhando a vida do usuário.

A maneira recomendada de interagir com o usuário é por meio de uma notificação, que é um alerta que fica na barra de status do Android, o qual chama a sua atenção. É simples de entender o conceito. Quando você recebe um email pela aplicação do Gmail, por acaso aparece alguma tela indesejada na tela do seu

celular? A resposta é não, pois o Gmail mostra uma agradável notificação na barra de status, e essa notificação você pode ver quando puder ou quiser.

Aproveitando a conversa, notificações é o assunto do próximo capítulo.

24.11 Links úteis

Neste capítulo, estudamos como interceptar mensagens de broadcast, que na prática são intents enviadas pela sua aplicação ou pelo próprio sistema operacional.

Seguem alguns links para continuar seus estudos.

- **API Reference – BroadcastReceiver**

<http://developer.android.com/reference/android/content/BroadcastReceiver.html>

- **Android Training – Reporting Work Status**

<https://developer.android.com/training/run-background-service/report-status.html>



CAPÍTULO 25

Notification

Uma notificação é uma mensagem especial que aparece na barra de status do Android para chamar a atenção do usuário.

Ao receber a notificação, o usuário pode decidir visualizar seu conteúdo ou simplesmente ignorar a mensagem. Quando o usuário clicar na notificação, será disparada uma *Intent* para iniciar uma *activity*, *broadcast receiver* ou *service*. O conteúdo dessa *intent*, ou seja, a sua ação e parâmetros, é você quem define. Novamente, as *intents* são o segredo de tudo.

25.1 Por que usar uma notificação para se comunicar com o usuário

Uma aplicação executando em segundo plano nunca deve exibir um alerta para o usuário, ou, pior ainda, abrir uma tela sem a sua permissão. Lembre-se de que o usuário pode estar lendo emails importantes, estar em uma chamada com um amigo, jogando ou executando qualquer outra aplicação.

Um exemplo de utilização de notificações seria uma aplicação de chat, na qual o usuário pode receber uma notificação quando chegar uma nova mensagem. Ao selecionar a notificação, o usuário pode decidir visualizar a mensagem ou não. Outro exemplo seria um serviço que executa em segundo plano, fazendo alguma atualização de informações. No momento em que a atualização terminar, a aplicação poderia informar o usuário utilizando uma notificação. A figura 25.1 mostra uma documentação oficial do Android que exibe vários tipos de notificações. A partir do Android 4.1, as notificações podem ser criadas com mais conteúdo, chamadas de **Big View Notifications**. Outra melhoria do Android 4.1 foi o suporte às ações diretamente nas notificações, que são botões customizados que aparecem na notificação.

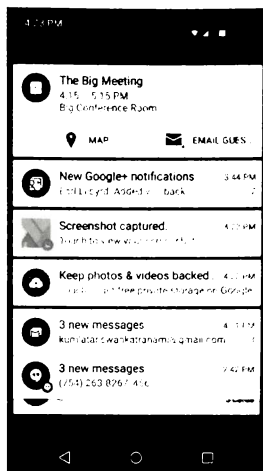


Figura 25.1 – Exemplo de notificações.

25.2 Criando uma notificação simples

A classe `Notification.Builder` contém vários métodos utilitários para configurar um objeto do tipo `Notification` e segue o famoso padrão de design Builder do Gof (Gang of Four). Caso você não conheça os padrões do Gof, recomendo fortemente que faça uma leitura complementar:

<http://pt.wikipedia.org/wiki/Builder>

http://pt.wikipedia.org/wiki/Design_Patterns

Como nem todas as funcionalidades das notificações existem em todas as versões do Android, foi criada a classe de compatibilidade `NotificationCompat.Builder`, que tem como objetivo esconder essa complexidade do desenvolvedor, pois utilizando a classe de compatibilidade as novas funcionalidades serão habilitadas se possível, ou serão ignoradas, tudo de forma transparente.

Para entendermos melhor como criar notificações, vamos criar um projeto chamado **HelloNotification**. Se preferir, abra o projeto de exemplo deste capítulo no Android Studio. Caso tenha criado o projeto, siga as próximas instruções, pois vamos construir várias notificações passo a passo. No arquivo de layout da activity adicione quatro botões, um para cada tipo de notificação que vamos criar.

 /res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="16dp" android:orientation="vertical">
    <TextView
        android:text="Exemplo de notificações"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <Button
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="Notificação Simples" android:onClick="onClickNotificacaoSimples" />
    <Button
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="Notificação Heads-up" android:onClick="onClickNotificacaoHeadsUp" />
    <Button
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="Notificação Big" android:onClick="onClickNotificacaoBig" />
    <Button
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="Notificação com Ação" android:onClick="onClickNotificacaoComAcao" />
</LinearLayout>
```

Repare que cada botão declara a tag `android:onClick`, portanto ao clicar neste botão o método configurado será executado na activity. Você pode adicionar os métodos manualmente no código da activity, ou utilizar o assistente do Android Studio para fazer isso automaticamente. A primeira forma de abrir o assistente é clicar na lâmpada amarela no código-fonte do arquivo de layout. Essa lâmpada é um indicador de que o assistente pode ajudá-lo com algo. Para exibi-la, fique com o cursor na linha em que você escreveu o código com a tag `android:onClick`. A segunda forma é utilizar a tecla de atalho **Alt+Enter** também nessa linha. Uma vez que o assistente for exibido, basta escolher a opção para criar o método.

Depois de criar os métodos, o código da activity deve ficar assim:

 MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_main);
    }
    public void onClickNotificacaoSimples(View view) { }
    public void onClickNotificacaoHeadsUp(View view) { }
```

```

    public void onClickNotificacaoBig(View view) { }
    public void onClickNotificacaoComAcao(View view) { }
}

```

A primeira notificação que vamos fazer é a mais simples e pode ser utilizada em todas as versões do Android, pois mostra apenas um ícone, título e mensagem. Para encapsular a criação das notificações, vamos criar a classe `NotificationUtil`, conforme o código demonstrado a seguir:



`NotificationUtil.java`

```

public class NotificationUtil {
    // Cria a PendingIntent para abrir a activity da intent
    private static PendingIntent getPendingIntent(Context context, Intent intent, int id) {
        TaskStackBuilder stackBuilder = TaskStackBuilder.create(context);
        // Esta linha mantém a activity pai na pilha de activities
        stackBuilder.addParentStack(intent.getComponent());
        // Configura a intent que vai abrir ao clicar na notificação
        stackBuilder.addNextIntent(intent);
        // Cria a PendingIntent e atualiza caso exista uma com o mesmo id
        PendingIntent p = stackBuilder.getPendingIntent(id, PendingIntent.FLAG_UPDATE_CURRENT);
        return p;
    }

    public static void create(Context context, Intent intent, String contentTitle,
        String contentText, int id) {
        // Cria a PendingIntent (contém a intent original)
        PendingIntent p = getPendingIntent(context, intent, id);
        // Cria a notificação
        NotificationCompat.Builder b = new NotificationCompat.Builder(context);
        b.setDefaults(Notification.DEFAULT_ALL); // Ativa configurações padrão
        b.setSmallIcon(R.drawable.ic_notification_icon); // Ícone
        b.setContentTitle(contentTitle); // Título
        b.setContentText(contentText); // Mensagem
        b.setContentIntent(p); // Intent que será chamada ao clicar na notificação.
        b.setAutoCancel(true); // Autocancela a notificação ao clicar nela
        NotificationManagerCompat nm = NotificationManagerCompat.from(context);
        // Neste caso a notificação será cancelada automaticamente quando o usuário clicar
        // Mas o id serve para cancelá-la manualmente se necessário.
        nm.notify(id, b.build());
    }

    public static void cancell(Context context, int id) {
        NotificationManagerCompat nm = NotificationManagerCompat.from(context);
        nm.cancel(id);
    }
}

```

```

    }
    public static void cancelAll(Context context) {
        NotificationManagerCompat nm = NotificationManagerCompat.from(context);
        nm.cancelAll();
    }
}

```

Nota: o método `setDefault(Notification.DEFAULT_ALL)` da classe `NotificationCompat.Builder` configura a notificação com um som padrão, faz vibrar o celular e ainda acende as luzes. Tecnicamente falando, a constante `DEFAULT_ALL` aplica todas as constantes `DEFAULT_SOUND`, `DEFAULT_VIBRATE` e `DEFAULT_LIGHTS`. Como nesse caso o celular vai vibrar, é preciso declarar a permissão `android.permission.VIBRATE` no arquivo de manifesto.

O código-fonte dessa classe está comentado, portanto leia-o com atenção. Basicamente estamos utilizando a classe `NotificationCompat.Builder` para criar a notificação com os parâmetros informados. Note que alguns parâmetros estão fixos, como o ícone da notificação, que deixei com o mesmo ícone do aplicativo. O importante é criar a intent e a `PendingIntent` que será disparada ao clicar na notificação. A `PendingIntent` é um tipo especial de intent que encasula uma intent real, a fim de ser disparada ao clicar na notificação.

Para concluir este primeiro exemplo, altere o método `onClickNotificacaoSimples(view)` para chamar o método `NotificationUtil.create(...)` e criar a notificação simples.



MainActivity.java

```

public class MainActivity extends AppCompatActivity {
    . . .
    public void onClickNotificacaoSimples(View view) {
        int id = 1;
        String contentTitle = "Nova mensagem";
        String contentText = "Você possui 3 novas mensagens";
        Intent intent = new Intent(this, MensagemActivity.class);
        intent.putExtra("msg", "Olá, Leitor, como vai?");
        NotificationUtil.create(this, intent, contentTitle, contentText, id);
    }
    . . .
}

```

Os outros métodos vamos deixar vazios por enquanto, pois construiremos os exemplos passo a passo. Veja que, na intent que foi criada para ser chamada ao clicar na notificação, utilizamos a classe `MensagemActivity` que ainda não existe. Inclusive estamos passando um parâmetro `"msg"` do tipo `String` para essa intent. Portanto,

utilize o wizard **New > Activity** do Android Studio para criar a activity `MensagemActivity` e configure a activity pai (parent activity) como a `MainActivity`. O código-fonte da classe `MensagemActivity` pode ser visualizado a seguir. No código estamos lendo o parâmetro "msg" que foi enviado pela intent para mostrar a mensagem no `TextView`.



MensagemActivity.java

```
public class MensagemActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_mensagem);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        String msg = getIntent().getStringExtra("msg");
        TextView text = (TextView) findViewById(R.id.text);
        text.setText(msg);
    }
}
```

No arquivo de layout, foi criado o `TextView` com o identificador `@+id/text`.



/res/layout/activity_mensagem.xml

```
<LinearLayout . . .>
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
</LinearLayout>
```

Para este exemplo funcionar corretamente, é importante que seja feita a configuração da activity `MensagemActivity` e de sua activity pai no arquivo de manifesto. Veja que configurei as tags `android:parentActivityName` e `<meta-data>` indicando que a activity pai na hierarquia é a `MainActivity`.



AndroidManifest.xml

```
<application . . .>
    <uses-permission android:name="android.permission.VIBRATE"/>
    <activity android:name=".MainActivity" . . . />
    <activity android:name=".MensagemActivity"
        android:parentActivityName=".MainActivity" >
        <meta-data android:name="android.support.PARENT_ACTIVITY"
            android:value=".MainActivity" />
    </activity>
</application>
```

Nota: lembre-se de adicionar a permissão `VIBRATE` no arquivo de manifesto. Isso é necessário, pois a notificação foi criada com o método `setDefaults(Notification.DEFAULT_ALL)`.

Depois dessa alteração, execute o projeto novamente e clique no primeiro botão. O resultado é a notificação exibida na figura 25.2. Ao clicar na notificação, a intent que foi configurada com a activity `MensagemActivity` é disparada, consequentemente esta tela é mostrada ao usuário. A mensagem **Olá, Leitor, como vai?** foi lida dos parâmetros da intent, apenas para demonstrar que podemos enviar os parâmetros.

O interessante das notificações é que elas podem executar mesmo com o aplicativo fechado. Portanto, depois de disparar a notificação, feche o aplicativo. Ao clicar na notificação, você verá que a activity `MensagemActivity` será chamada. Agora faça o teste, e clique no botão voltar. O resultado é que a `MainActivity` foi exibida, certo? Mas se a aplicação estava fechada, e chamamos a `MensagemActivity` pela intent, como fizemos para colocar a `MainActivity` na base da pilha? Isso foi feito no método `getPendingIntent()` da classe `NotificationUtil`, com a ajuda da classe `TaskStackBuilder`. A linha de código a seguir adiciona a activity pai na pilha. Se você comentar esta linha, somente a classe `MensagemActivity` será adicionada à pilha.

```
// Esta linha mantém a activity pai na pilha de activities  
stackBuilder.addParentStack(intent.getComponent());
```

Essa é uma decisão de design importante, pois na maioria das vezes, ao abrir uma activity, também precisamos carregar a sua activity pai.

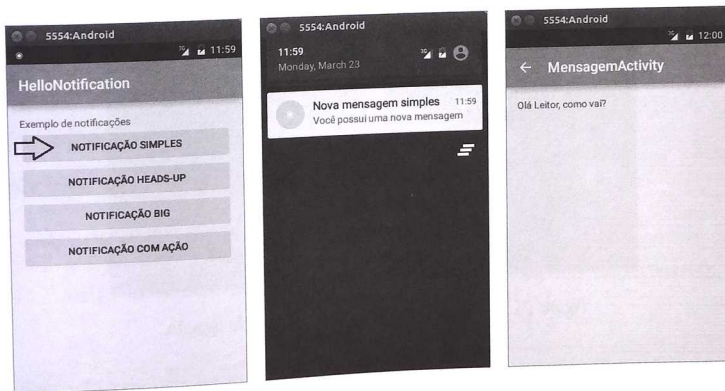


Figura 25.2 – Notificação simples.

Dica: o ícone de notificação do projeto de exemplo deste capítulo foi criado com o wizard **New Image Asset** do Android Studio. No wizard, selecione o tipo **Notification** e clique em **Clipart** para escolher algum modelo de ícone. O Android Studio vai criar o modelo de ícone adequado para cada versão da plataforma do Android. No Android 5.0, devido ao Material Design, os ícones de notificações devem ser brancos com fundo transparente, pois o sistema vai desenhá-los com um fundo circular.

25.3 Heads-up notifications

No Android 5.0, foram criadas as notificações conhecidas como **heads-up notifications**, que aparecem no topo da tela por cima de tudo. Um exemplo é a notificação que aparece quando você recebe uma ligação telefônica, a qual o usuário pode escolher atender ou não.

Antigamente, o próprio aplicativo nativo da ligação mostrava uma tela grande para o usuário escolher se ele deseja atender ou não a ligação (Figura 25.3). Com as **heads-up notifications**, a notificação ficou mais elegante e sutil, e o usuário pode continuar vendo a tela em que estava.

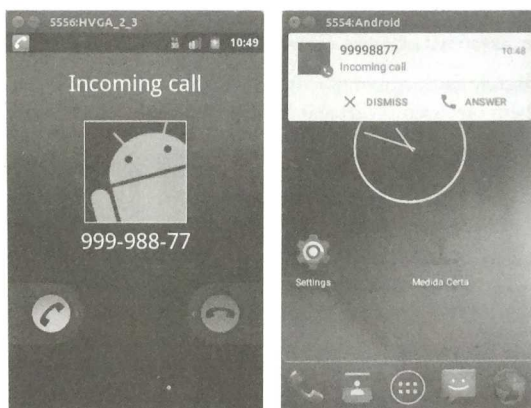


Figura 25.3 – Heads-up notification atendendo ligação.

Dica: uma heads-up notification pode ser removida fazendo o gesto de swipe lateral, ou seja, segure-a e arremesse para a direita, jogando a notificação para fora da tela.

Para criar uma **heads-up notification**, basta chamar o método `setFullScreenIntent(PendingIntent, boolean)` da classe `NotificationCompat.Builder`. Para testar o conceito, crie o método `createHeadsUpNotification(...)` na classe `NotificationUtil`.



NotificationUtil.java

```
public class NotificationUtil {  
    . . .  
    public static void createHeadsUpNotification (. . .) {  
        PendingIntent p = . . . // Tudo igual antes...  
        NotificationCompat.Builder b = . . .;  
        // Cor da notificação Android 5.0  
        b.setColor(Color.BLUE);  
        // Heads-up notification  
        b.setFullScreenIntent(p, false);  
        . . .  
        nm.notify(id, b.build());  
    }  
}
```

Além do método `setFullScreenIntent(PendingIntent, boolean)`, você deve ter reparado que também chamei o método `setColor(int)`, que recebe o RGB de uma cor. No Material Design, o ícone da notificação deve ser branco com fundo transparente, pois será desenhado de forma circular pelo sistema. O método `setColor(int)` justamente pinta o fundo transparente com a cor informada. Mais uma vez, atente-se para a importância de utilizar a classe de compatibilidade `NotificationCompat`, pois ela ativa esses recursos nas novas versões do Android, ou simplesmente os oculta nas versões antigas.

Para testar a **heads-up notification**, atualize o código da classe `MainActivity`.



MainActivity.java

```
public class MainActivity extends AppCompatActivity {  
    . . .  
    public void onClickNotificacaoHeadsUp(View view) {  
        // Crie a intent como no anterior ...  
        NotificationUtil.createHeadsUpNotification(this, intent, contentTitle, contentText, id);  
    }  
}
```

A figura 254 mostra o resultado da **heads-up notification**.



Figura 25.4 – Heads-up notification.

25.4 Notificações na tela de bloqueio

No Android 5.0, as notificações aparecem na tela de bloqueio por padrão, conforme a figura 25.5. Para testar esse comportamento, implemente o método `onClickNotificacaoWithDelay()`. Esse método cria a notificação como antes, porém com um atraso de dois segundos, tempo suficiente para você colocar a tela do celular no modo bloqueio.



MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    ...
    public void onClickNotificacaoWithDelay(View view) {
        view.postDelayed(new Runnable() {
            @Override
            public void run() {
                ... // Aqui é igual ao código anterior...
                NotificationUtil.createHeadsUpNotification(getBaseContext(),
                    intent, contentTitle, contentText, id);
                Toast.makeText(getBaseContext(), "Coloque o cell na lock-screen",
                    Toast.LENGTH_SHORT).show();
            }
        }, 2000); // Delay de dois segundos
    }
}
```

Para customizar o modo de visibilidade da notificação na tela de bloqueio, utilize o método `setVisibility(int)` da classe `NotificationCompat.Builder`, informando uma das seguintes constantes:

- `NotificationCompat.VISIBILITY_PUBLIC` – Mostra a notificação completa na tela de bloqueio.
- `NotificationCompat.VISIBILITY_PRIVATE` – Não mostra o conteúdo da notificação na tela de bloqueio.
- `NotificationCompat.VISIBILITY_SECRET` – Não mostra a notificação completa na tela de bloqueio.

O lado direito da figura 25.5 mostra uma notificação criada com a visibilidade `VISIBILITY_PRIVATE`, portanto o conteúdo ficou escondido.

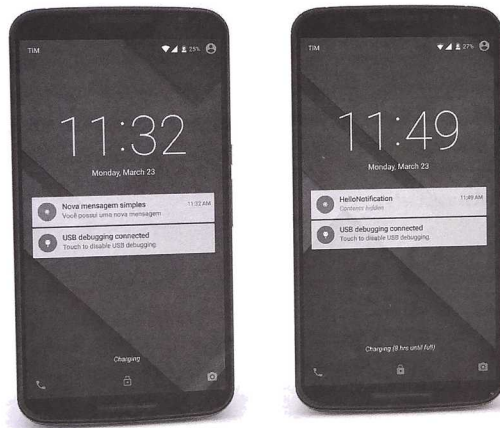


Figura 25.5 – Notificação na tela de bloqueio.

Atenção: no Android 5.0, entre nas configurações do sistema na opção `Sound & Configuration > Notification > When device is locked` para você configurar como deseja que as notificações na tela de bloqueio se comportem. Se o usuário tiver uma tela de bloqueio protegida (ex.: precisa digitar um PIN) e selecionar a opção `Hide sensitive notification content`, neste caso sim as constantes de visibilidade serão aplicadas e surtirão efeito. Se seu celular estiver em português, procure pelas opções de Som e Notificação > Não mostrar notificações.

25.5 Criando uma notificação grande (big view notifications)

A partir do Android 4.1, foi criado o conceito de big view notifications, que são notificações que podem ocupar um espaço maior na tela, conter um texto com várias linhas, ou até ações customizadas, como os botões de atender a ligação que vimos no exemplo anterior.

Para demonstrar como criar uma notificação expandida com várias linhas, crie o método `createBig(...)` na classe `NotificationUtil`.



NotificationUtil.java

```
public class NotificationUtil {
    . . .
    public static void createBig(Context context, Intent intent, String contentTitle,
        String contentText, List<String> lines, int id) {
        PendingIntent p = getPendingIntent(context, intent, id);
        // Configura o estilo Inbox
        int size = lines.size();
        NotificationCompat.InboxStyle inboxStyle = new NotificationCompat.InboxStyle();
        inboxStyle.setBigContentTitle(contentTitle);
        for (String s: lines) {
            inboxStyle.addLine(s);
        }
        inboxStyle.setSummaryText(contentText);
        // Cria a notificação
        NotificationCompat.Builder b = new NotificationCompat.Builder(context);
        b.setDefaults(Notification.DEFAULT_ALL); // Ativa configurações padrão
        b.setSmallIcon(R.drawable.ic_notification_icon); // Ícone
        b.setContentTitle(contentTitle); // Título
        b.setContentText(contentText); // Mensagem
        b.setContentIntent(p); // Intent que será chamada ao clicar na notificação.
        b.setAutoCancel(true); // Autocancela a notificação ao clicar nela
        b.setNumber(size); // Número para aparecer na notificação
        b.setStyle(inboxStyle); // Estilo customizado
        NotificationManagerCompat nm = NotificationManagerCompat.from(context);
        nm.notify(id, b.build());
    }
    . . .
}
```

No código foi utilizada a classe `InboxStyle` para configurar o estilo da notificação e adicionar várias linhas na mensagem. Agora, atualize o código da classe `MainActivity` para criar a notificação grande.



MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    ...
    public void onClickNotificacaoBig(View view) {
        int id = 2;
        String contentTitle = "Nova mensagem";
        String contentText = "Você possui 3 novas mensagens";
        Intent intent = new Intent(this, MensagemActivity.class);
        intent.putExtra("msg", "Olá, Leitor, como vai?");
        List<String> lines = new ArrayList<>();
        lines.add("Mensagem 1");
        lines.add("Mensagem 2");
        lines.add("Mensagem 3");
        NotificationUtil.createBig(this, intent, contentTitle, contentText, lines, id);
    }
}
```

Neste exemplo, criamos uma lista com três mensagens, e ao clicar no segundo botão o resultado será como a figura 25.6. Repare que no canto direito inferior da notificação apareceu o número 3, pois no código utilizei o método `setNumber(int)` da classe `NotificationCompat.Builder`.

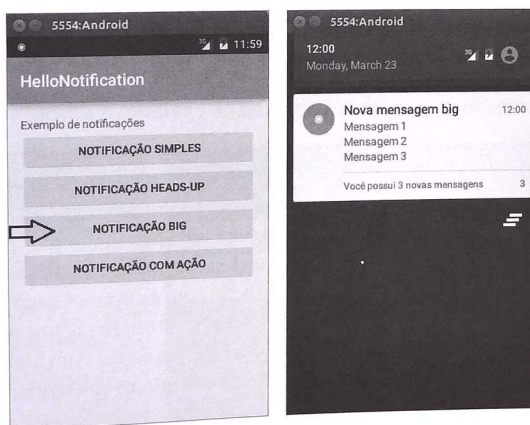


Figura 25.6 – Exemplo com notificação expandida.

25.6 Criando uma notificação com ações

Para criar uma notificação com ação, basta chamar o método `builder.addAction(icone, string, intent)` configurando o ícone da ação, o texto e a intent que será disparada ao clicar na ação.

Para o próximo exemplo, crie o método `createWithAction(...)` na classe `NotificationUtil`. Repare que resumi o código, pois a forma de criar a notificação é igual ao exemplo da notificação simples, com exceção de que estou chamando o método `builder.addAction(icone, string, intent)`.



NotificationUtil.java

```
public class NotificationUtil {
    public static final String ACTION_VISUALIZAR = "ACTION_VISUALIZAR";
    . . .
    public static void createWithAction(Context context, Intent intent, String contentTitle,
        String contentText, int id) {
        // Crie a NotificationCompat.Builder normalmente aqui...
        // Ação customizada (deixei a mesma intent para as duas ações)
        PendingIntent actionIntent = PendingIntent.getBroadcast(
            context, 0, new Intent(ACTION_VISUALIZAR), 0);
        b.addAction(R.drawable.ic_acao_pause, "Pause", actionIntent);
        b.addAction(R.drawable.ic_acao_play, "Play", actionIntent);
        NotificationManagerCompat nm = NotificationManagerCompat.from(context);
        nm.notify(id, b.build());
    }
}
```

Para o código compilar, adicione os ícones pause e play na pasta `/res/drawable`. Uma dica interessante é criar os ícones com o wizard **New Image Asset**. No wizard, selecione o tipo **Notification** e clique em **Clipart** para escolher algum modelo de ícone. Foi assim que criei os ícones de play e pause deste exemplo. Para finalizar, atualize o código da `MainActivity` para criar a notificação com as ações. Como uma intent será disparada ao clicar na ação, precisamos registrar um broadcast receiver para receber o evento. Note que estou disparando a mesma intent para ambas as ações para simplificar o código, mas na prática teríamos intents diferentes.



MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    . . .
    private BroadcastReceiver customActionReceiver = new BroadcastReceiver() {
        @Override
```

```

public void onReceive(Context context, Intent intent) {
    Toast.makeText(context, "CLICOU NA AÇÃO!", Toast.LENGTH_SHORT).show();
    NotificationUtil.cancel(context, 3);
}

@Override
protected void onResume() {
    super.onResume();
    registerReceiver(customActionReceiver, new IntentFilter(NotificationUtil.ACTION_VISUALIZAR));
}

@Override
protected void onPause() {
    super.onPause();
    unregisterReceiver(customActionReceiver);
}

...

public void onClickNotificacaoComAcao (View view) {
    int id = 3;
    Intent intent = new Intent(this, MensagemActivity.class);
    intent.putExtra("msg", "Música legal.");
    String contentTitle = "Nome da música";
    String contentText = "Nome do artista";
    NotificationUtil.createWithAction(this, intent, contentTitle, contentText, id);
}
}

```

A figura 25.7 mostra o resultado ao criar uma notificação com ação.

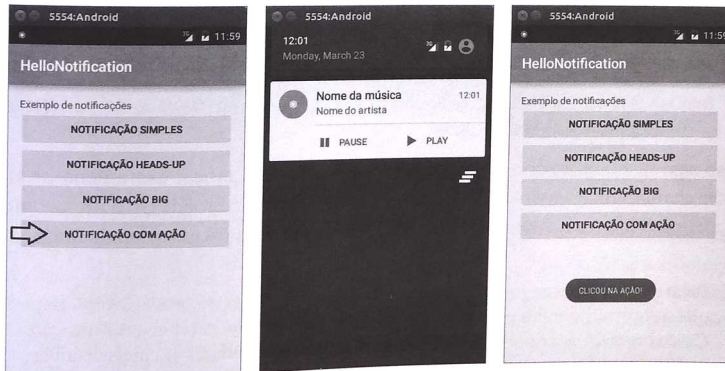


Figura 25.7 – Exemplo de notificação com ação.

Ao clicar nos botões, uma intent é disparada. A terceira parte da figura mostra o alerta informando que o broadcast receiver recebeu a intent. Lembre-se de que usei a mesma intent e receiver para ambas as ações para simplificar o código, mas na prática você teria uma intent e receiver para a ação **pause** e outra para o **play**.

25.7 Cancelando uma notificação

Para cancelar uma notificação, ou seja, removê-la da barra de status, podemos utilizar os seguintes métodos da classe `NotificationManager`:

- `cancel(int id)` – Cancela a notificação utilizando o id fornecido.
- `cancelAll()` – Cancela todas as notificações do aplicativo.

Outra forma é chamar o método `setAutoCancel(boolean)` da classe `NotificationCompat.Builder` no momento de criar a notificação, assim ela será automaticamente cancelada quando o usuário abri-la.

25.8 Mais informações sobre a classe `PendingIntent`

Na classe `NotificationUtil` criamos um objeto do tipo `PendingIntent` com a classe `TaskStackBuilder`, a qual consegue ter um maior controle para criar a pilha de activities, sendo que naquele exemplo a pilha era formada por: `MainActivity > MensagemActivity`.

Mas a forma clássica de criar uma `PendingIntent` para chamar uma activity é assim:

```
PendingIntent p = PendingIntent.getActivity(this, 0, new Intent(this, NomeDaActivity.class),
    PendingIntent.FLAG_UPDATE_CURRENT);
```

E para criar uma `PendingIntent` que vai disparar um broadcast para um receiver é assim:

```
PendingIntent p = PendingIntent.getBroadcast(this, 0, new Intent("ABRIR_APLICACAO_TESTE"),
    PendingIntent.FLAG_UPDATE_CURRENT);
```

Isso significa que, ao criar a `PendingIntent`, é preciso especificar exatamente se ela deve chamar uma activity ou disparar uma mensagem de broadcast. É importante você guardar bem na memória essa informação, pois ela será útil mais tarde.

Dica: recomendo sempre criar a `PendingIntent` com a flag `FLAG_UPDATE_CURRENT`, para que o Android sempre entregue uma intent atualizada ao receber a notificação. Caso contrário, você pode receber mensagens desatualizadas. Para mais detalhes, verifique a documentação oficial.

25.9 Exemplo com notificação e BroadcastReceiver

No capítulo anterior, criamos um projeto chamado `HelloReceiver` que mostrou como enviar uma intent por broadcast e interceptar a mensagem utilizando um broadcast receiver.

Para exercitar o conceito de notificações, vamos aprimorar esse exemplo e mostrar uma notificação quando o receiver receber a mensagem. Para continuar, copie o projeto `HelloReceiver` do capítulo anterior e renomeie para `HelloReceiverNotification`.

Este projeto já declara um receiver estático, que está configurado no `AndroidManifest.xml` para interceptar as mensagens com ação `BINGO`. Como queremos criar uma notificação, adicione a seguinte classe ao projeto:



`NotificationUtil.java`

```
public class NotificationUtil {
    public static void notify(Context context, int id, Intent intent, String contentTitle,
        String contentText) {
        NotificationManager manager =
            (NotificationManager) context.getSystemService(Context.NOTIFICATION_SERVICE);
        // Intent para disparar o broadcast
        PendingIntent p = PendingIntent.getActivity(context, 0, intent,
            PendingIntent.FLAG_UPDATE_CURRENT);
        // Cria a notification
        NotificationCompat.Builder builder = new NotificationCompat.Builder(context)
            .setContentIntent(p)
            .setContentTitle(contentTitle)
            .setContentText(contentText)
            .setSmallIcon(R.drawable.ic_notification_icon)
            .setAutoCancel(true);
        // Dispara a notification
        Notification n = builder.build();
        manager.notify(id, n);
    }
}
```

Na sequência, vamos atualizar o código da classe `HelloReceiver` para mostrar uma notificação ao receber uma mensagem de broadcast. Observe que a intent do receiver recebe um parâmetro `"msg"` do tipo `string` e está repassando esse parâmetro para a intent que será disparada ao clicar na notificação. A intent da notificação vai abrir a `MainActivity`.



HelloReceiver.java

```
public class HelloReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d("livroandroid", "HelloReceiver !!!");
        // Recebeu a mensagem de broadcast.
        Intent notifIntent = new Intent(context, MainActivity.class);
        String msg = intent.getStringExtra("msg");
        notifIntent.putExtra("msg", msg);
        // Cria a notificação
        NotificationUtil.notify(context, 1, notifIntent, "Texto digitado", msg);
    }
}
```

No arquivo de layout da activity, adicione um campo de texto para o usuário digitar a mensagem.

```
// arquivo /res/layout/activity_main.xml
<EditText android:id="@+id/text"
    android:layout_width="match_parent" android:layout_height="wrap_content" />
```

Por último, no código da activity vamos disparar a mensagem de broadcast com a ação BINGO. A diferença para o exemplo do capítulo anterior é que estamos lendo o texto digitado no EditText e enviado por parâmetro com o chave "msg". Lembre-se de que o receiver está mapeado no *AndroidManifest.xml* para interceptar a ação BINGO.



MainActivity.java

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        findViewById(R.id.btEnviar).setOnClickListener(this);
        readMsg(getIntent());
    }
    @Override
    protected void onNewIntent(Intent intent) {
        super.onNewIntent(intent);
        readMsg(intent);
    }
    private void readMsg(Intent intent) {
        String msg = intent.getStringExtra("msg");
        if(msg != null) {
```



```

        Toast.makeText(this, "Você digitou: " + msg, Toast.LENGTH_SHORT).show();
    } else {
        Toast.makeText(this, "Extras: " + intent.getExtras(), Toast.LENGTH_SHORT).show();
    }
}

@Override
public void onClick(View v) {
    EditText text = (EditText) findViewById(R.id.text);
    String msg = text.getText().toString();
    Intent intent = new Intent(new Intent("BINGO"));
    intent.putExtra("msg", msg);
    sendBroadcast(intent);
    Toast.makeText(this, "Intent enviada!", Toast.LENGTH_SHORT).show();
}
}

```

A figura 25.8 mostra o resultado desse exemplo. O fluxo é simples: o receiver vai receber a mensagem e mostrar a notificação. Quando o usuário clicar na notificação, a MainActivity será chamada e neste momento estou lendo o parâmetro "msg" que chegou da intent.

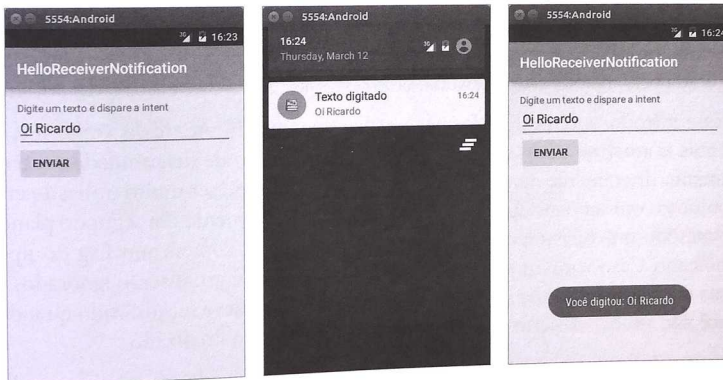


Figura 25.8 – Broadcast receiver com notificação.

Este exemplo tem uma funcionalidade interessante, pois veja que no código da classe MainActivity foi adicionado o método `onNewIntent(intent)`. Por padrão, sempre que uma intent é chamada, uma nova activity é criada. Mas neste caso eu não quero esse comportamento, pois precisamos garantir que sempre exista apenas uma MainActivity viva na pilha de activities. Para isso, configure a tag `android:launchMode="singleTop"` no `AndroidManifest.xml`.

```
// AndroidManifest.xml
<activity android:name=".MainActivity" android:launchMode="singleTop" ... />
```

Isso mantém apenas uma `MainActivity` na pilha. Caso a `activity` não exista, ela será criada; caso contrário, a sua instância será preservada em memória e o método `onNewIntent(intent)` será chamado. Exatamente por isso o código faz a leitura do parâmetro `"msg"` em dois lugares diferentes: nos métodos `onCreate(bundle)` e `onNewIntent(intent)`.

Dica: entender esse exemplo é fundamental para dominar o desenvolvimento para Android. Um `receiver` pode interceptar mensagens de broadcast, sejam elas da sua aplicação ou de sistema. Porém, veja que o `receiver` não chamou a `MainActivity`, pois isso é agressivo demais. O que fizemos foi mostrar uma notificação para o usuário. Quando o usuário clicar na notificação, é disparada uma nova `intent` com o objetivo de abrir a `activity`, pois agora podemos fazer isso, e foi o usuário quem executou a ação.

25.10 Mostrando uma barra de progresso na notificação

Uma funcionalidade bem interessante que foi adicionada ao Android 4.0 é a opção de exibir uma barra de progresso na notificação. Essa barra pode ser facilmente configurada pelo método `setProgress(max, progresso, infinito)` da classe `Notification.Builder`.

Nesse método, você pode informar o valor máximo do progresso, como 100, e depois ir atualizando os valores para informar o status de determinado processamento diretamente na notificação. Esse conceito pode ser muito utilizado em conjunto com serviços que estão fazendo um processamento em segundo plano. O método `setProgress(max, progresso, infinito)` também contém um flag do tipo booleano. Caso você informe `true`, os valores máximo e atual serão ignorados e uma animação contínua será exibida. Esse parâmetro deve ser utilizado quando você não souber ao certo quando o processamento será finalizado.

A figura 25.9 mostra como seria a notificação com a barra de progresso normal e com o flag infinito para `true`. Na figura da esquerda, o valor informado foi 50% com o seguinte código:

```
builder.setProgress(100, 50, false);
```

Na figura da direita, o valor informado está infinito e não determinado com o seguinte código:

```
builder.setProgress(0, 0, true);
```

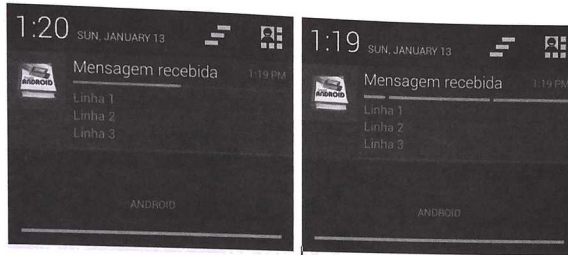


Figura 25.9 – Notificação com barra de progresso.

25.11 Links úteis

Notificações representam um mecanismo importante da plataforma do Android para se comunicar com o usuário. O assunto parece simples, mas recomendo fortemente que você complemente seus estudos com a documentação oficial.

- **Android API Guides – Notifications**

<http://developer.android.com/guide/topics/ui/notifiers/notifications.html>

- **Android Training – Notifying the User**

<http://developer.android.com/training/notify-user/index.html>

- **Android Design – Notifications**

<http://developer.android.com/design/patterns/notifications.html>

- **Android DevBytes: Notificações no L Developer Preview [Portuguese]**

<https://www.youtube.com/watch?v=EHTU5CxhoZ4&list=PLiGzvgaA5Gmg5EovEGdPo5SaFv74INdHx>



CAPÍTULO 26

AlarmManager

A classe `AlarmManager` permite agendar uma aplicação para ser executada em determinado momento no futuro. Para isso, uma `Intent` é utilizada para enviar uma mensagem ao sistema operacional na data e na hora desejadas.

Esse processo é conhecido como agendar um alarme e pode ser usado para simplesmente agendar uma aplicação para executar todos os dias à meia-noite, ou para ficar repetindo determinado processo a cada hora.

26.1 Por que utilizar um alarme (agendar uma tarefa)

Imagine que você precisa executar uma aplicação a cada 30 minutos, ou a cada hora, ou executar algo todo dia ao meio-dia. Antes que sua criatividade comece a ganhar asas, explicaremos a maneira correta de fazer isso, que é utilizando um alarme.

Um alarme é criado utilizando a classe `android.app.AlarmManager` e define a data e a hora em que uma `Intent` deve ser disparada ao sistema operacional. Depois, qualquer aplicação pode interceptar essa mensagem/intent para iniciar uma aplicação ou serviço.

Na prática, as intents dos alarmes são interceptadas por broadcast receivers, que por sua vez decidem o que fazer. Talvez o receiver mostre uma notificação ou simplesmente seja disparado um processo em segundo plano (veja o capítulo 27, sobre a classe `Service`).

A vantagem de um alarme é que depois que ele é ativado você pode até esquecê-lo. Mesmo se o celular ficar inativo (entrar em modo de espera), o alarme continuará lá, vivo e pronto para disparar na data e na hora para as quais foi configurado.

26.2 Método da classe AlarmManager

Para agendar um alarme, utiliza-se a classe `android.app.AlarmManager`. Na lista a seguir, podemos ver os métodos mais importantes dessa classe.

`set(int tipo, long triggerAtMillis, PendingIntent intent)`

Método usado para agendar o alarme, informando a data e hora em que ele deve ser disparado, bem como a intent que contém a mensagem a ser enviada. O parâmetro do tipo `long` é o tempo corrido em milissegundos e pode ser extraído a partir de objetos do tipo `java.util.Date` ou `java.util.Calendar`.

`setRepeating(int tipo, long triggerAtMillis, long intervalMillis, PendingIntent intent)`

Idêntico ao anterior, mas neste caso o alarme repetirá continuamente de tempos em tempos, especificado pelo parâmetro intervalo em milissegundos. O alarme será repetido até que o método `cancel(intent)` seja chamado. Isso é útil, por exemplo, se você estiver desenvolvendo um aplicativo de dieta balanceada e precisa lembrar o usuário de comer de três em três horas.

`setInexactRepeating(int tipo, long triggerAtMillis, long intervalMillis, PendingIntent intent)`

Este método é uma alternativa ao `setRepeating(...)` e segundo a documentação do Android ele economiza recursos. Caso mais de uma aplicação agende um alarme na mesma hora, o Android consegue fazer otimizações e disparar o mesmo alarme para ambas as aplicações.

`cancel(PendingIntent intent)`

Cancela o alarme, baseado na intent fornecida, que deve ser a mesma utilizada para iniciar o alarme.

Os métodos `set(tipo,dataHora,intent)` e `setRepeating(tipo,dataHora,intervalo,intent)` fazem a mesma coisa, mas o segundo ainda é capaz de repetir o alarme automaticamente durante o tempo estipulado. Independentemente de como o alarme é criado, basta chamar o método `cancel(intent)` para cancelá-lo.

Para demonstrar como criar um alarme, segue um trecho de código que dispara o alarme exatamente às 9:00 am e repete a cada 30 minutos:

```
private AlarmManager alarmMgr;  
private PendingIntent alarmIntent;  
...  
// Lê o serviço do AlarmManager
```



```

alarmMgr = (AlarmManager) context.getSystemService(Context.ALARM_SERVICE);
// Prepara a intent que vai receber o alarme
Intent intent = new Intent(context, MeuBroadcastReceiverAqui.class);
alarmIntent = PendingIntent.getBroadcast(context, 0, intent, 0);
// Calcula a hora para 9:00 am.
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.set(Calendar.HOUR_OF_DAY, 9);
calendar.set(Calendar.MINUTE, 30);
// Agenda a cada 30 minutos (precisa informar o tempo em milissegundos)
alarmMgr.setRepeating(AlarmManager.RTC_WAKEUP, calendar.getTimeInMillis(),
    1000 * 60 * 30, alarmIntent);

```

Repare que ao chamar o método `setRepeating(...)` informamos a constante `AlarmManager.RTC_WAKEUP`, sendo que existem quatro constantes.

ELAPSED_REALTIME

Este tipo de alarme é útil se você deseja dispará-lo a partir do tempo atual. Não necessariamente em uma hora específica como 9:00, 9:30 ou 10:00. Por exemplo, se for 9:10, e você disparar o alarme para daqui a 30 minutos, ele executaria às 9:40. Neste caso, você não está interessado na hora exata que o alarme vai executar, e sim em que ele execute daqui a 30 minutos ou uma hora.

ELAPSED_REALTIME_WAKEUP

Idem à constante anterior, mas neste caso a CPU é acordada para receber o alarme, mesmo se o dispositivo estiver no modo de espera. Por padrão, os alarmes só são recebidos se estiverem ativados.

RTC

Dispara um alarme em uma data e hora específica que pode ser configurada no código.

RTC_WAKEUP

Idem à constante anterior, porém acorda a CPU ao receber o alarme.

Baseado nas explicações acima, eu geralmente utilizo uma data específica e prefiro utilizar as constantes `RTC` e `RTC_WAKEUP`.

26.3 Agendando um alarme

Para demonstrar o uso dos alarmes, vamos criar um projeto chamado **HelloAlarme**, ou se preferir abra o projeto de exemplo deste capítulo no Android Studio.

Digamos que precisamos construir um aplicativo de dieta balanceada, e nosso objetivo é mostrar alarmes para o usuário a fim de lembrá-lo de comer a cada três horas. Como três horas é muito tempo para testar, vamos fazer a mesma coisa, porém vamos disparar os alarmes a cada 30 segundos.

Nota: quando falamos em executar ou agendar um alarme, estamos agendando uma intent para ser disparada em determinadas data e hora. O alarme é agendado no sistema operacional, portanto sua aplicação vai receber a intent mesmo se estiver fechada.

A seguir, podemos verificar o arquivo *AndroidManifest.xml* do projeto:



AndroidManifest.xml

```
<manifest ... package="br.com.livroandroid.helloalarme" >
    <application ... >
        <activity android:name=".MainActivity" android:launchMode="singleTop" ... />
        <receiver android:name=".LembreDeComerReceiver">
            <intent-filter>
                <action android:name="br.com.livroandroid.helloalarme.LEMBREME_DE_COMER" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

O arquivo *AndroidManifest.xml* configurou a *MainActivity* com o item `android:launchMode="singleTop"`, pois vamos criar uma notificação e queremos manter apenas uma instância dessa activity na pilha. Também foi configurado um broadcast receiver para a ação `LEMBREME_DE_COMER`, sendo que essa classe pode ser visualizada a seguir.



LembreDeComerReceiver.java

```
import livroandroid.lib.utils.NotificationUtil;

public class LembreDeComerReceiver extends BroadcastReceiver {
    private static final String TAG = "livroandroid";
    public static final String ACTION = "br.com.livroandroid.helloalarme.LEMBREME_DE_COMER";
}
```

```

@Override
public void onReceive(Context context, Intent intent) {
    Log.d(TAG, "Você precisa comer: " + new Date());
    Intent notifIntent = new Intent(context, MainActivity.class);
    NotificationUtil.create(context, 1, notifIntent, R.mipmap.ic_launcher,
        "Hora de comer algo...", "Que tal uma fruta?");
}
}

```

O receiver vai interceptar a intent que será disparada pelo alarme e vai mostrar uma notificação. A classe `NotiticationUtil` você pode criar como foi feito no projeto anterior, ou se preferir utilize a classe `livroandroid.lib.utils.NotificationUtil` da biblioteca `android-utils`.

Para disparar os alarmes, vamos criar a classe `AlarmUtil`. O objetivo desta classe é apenas encapsular o acesso à classe `AlarmManager` do Android.



AlarmUtil.java

```

public class AlarmUtil {
    private static final String TAG = "livroandroid";
    // Agenda o alarme na data/hora informado.
    public static void schedule(Context context, Intent intent, long triggerAtMillis) {
        PendingIntent p = PendingIntent.getBroadcast(context, 1, intent,
            PendingIntent.FLAG_UPDATE_CURRENT);
        AlarmManager alarme = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
        alarme.set(AlarmManager.RTC_WAKEUP, triggerAtMillis, p);
        Log.d("livroandroid-alarm", "Alarme agendado com sucesso.");
    }
    // Agenda o alarme com a opção de repetir
    public static void scheduleRepeat(Context context, Intent intent, long triggerAtMillis,
        long intervalMillis) {
        PendingIntent p = PendingIntent.getBroadcast(context, 1, intent,
            PendingIntent.FLAG_UPDATE_CURRENT);
        AlarmManager alarme = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
        alarme.setInexactRepeating(AlarmManager.RTC_WAKEUP, triggerAtMillis, intervalMillis, p);
        Log.d("livroandroid-alarm", "Alarme agendado com sucesso com repeat.");
    }
    // Cancela o alarme
    public static void cancel(Context context, Intent intent) {
        AlarmManager alarme = (AlarmManager)context.getSystemService(Context.ALARM_SERVICE);
        PendingIntent p = PendingIntent.getBroadcast(context, 1, intent,
            PendingIntent.FLAG_UPDATE_CURRENT);
    }
}

```

```

        alarme.cancel(p);
        Log.d("livroandroid-alarm", "Alarme cancelado com sucesso.");
    }
}

```

Feito isso, no layout da activity vamos adicionar três botões. O primeiro vai agendar um alarme para daqui a cinco segundos. O segundo vai fazer a mesma coisa, mas vai repetir o alarme a cada 30 segundos. E o terceiro botão vai cancelar o alarme.

 /res/layout/activity_main.xml

```

<LinearLayout . . .>
    <TextView android:text="Agendar o alarme" . . . />
    <Button android:text="Agendar para 5 segundos" . . .
        android:onClick="onClickAgendar"/>
    <Button android:text="Agendar e repetir a cada 30 seg" . . .
        android:onClick="onClickAgendarComRepeat" />
    <Button android:text="Cancelar"
        android:onClick="onClickCancelar"/>
</LinearLayout>

```

Por último, no código da activity vamos adicionar os três métodos para tratar os eventos.

 MainActivity.java

```

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    // Data/Tempo para agendar o alarme
    public long getTime() {
        Calendar c = Calendar.getInstance();
        c.setTimeInMillis(System.currentTimeMillis());
        c.add(Calendar.SECOND, 5); // 5 segundos
        long time = c.getTimeInMillis();
        return time;
    }
    public void onClickAgendar(View view) {
        Intent intent = new Intent(LembreDeComerReceiver.ACTION);
        // Agenda para daqui a 5 segundos
        AlarmUtil.schedule(this, intent, getTime());
    }
}

```

```

        Toast.makeText(this, "Alarme agendado.", Toast.LENGTH_SHORT).show();
    }
    public void onClickAgendarComRepeat(View view) {
        Intent intent = new Intent(LembreDeComerReceiver.ACTION);
        // Agenda para daqui a 5 segundos, repete a cada 30 segundos
        AlarmUtil.scheduleRepeat(this, intent, getTime(), 30 * 1000);
        Toast.makeText(this, "Alarme agendado com repetir.", Toast.LENGTH_SHORT).show();
    }
    public void onClickCancelar(View view) {
        Intent intent = new Intent(LembreDeComerReceiver.ACTION);
        AlarmUtil.cancel(this, intent);
        Toast.makeText(this, "Alarme cancelado", Toast.LENGTH_SHORT).show();
    }
}

```

Ao executar o exemplo e clicar no botão para agendar o alarme, a intent será disparada depois de cinco segundos. O receiver vai interceptar essa intent e mostrar uma notificação conforme a figura 26.1. O segundo botão também agenda o alarme para daqui cinco segundos, a diferença é que o alarme ficará repetindo a cada 30 segundos.

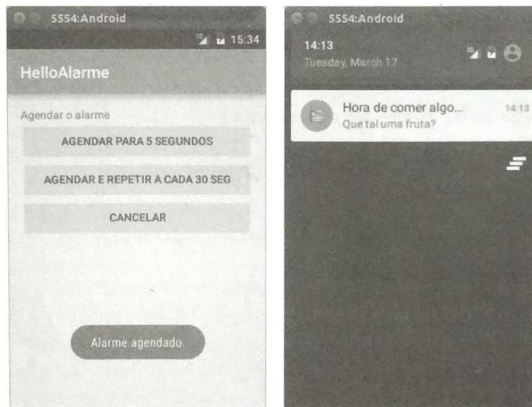


Figura 26.1 – Alarme com notificação.

26.4 Repetindo o alarme

No código que fizemos para repetir o alarme a cada 30 segundos, foi passado o tempo em milissegundos, portanto fizemos a conta $30 * 1.000$.

Caso você prefira utilizar constantes para fazer essa conta, a classe `AlarmManager` contém algumas, conforme demonstrado a seguir:

```
long INTERVAL_FIFTEEN_MINUTES = 15 * 60 * 1000;
long INTERVAL_HALF_HOUR = 2 * INTERVAL_FIFTEEN_MINUTES;
INTERVAL_HOUR = 2 * INTERVAL_HALF_HOUR;
INTERVAL_HALF_DAY = 12 * INTERVAL_HOUR;
INTERVAL_DAY = 2 * INTERVAL_HALF_DAY;
```

Portanto, para agendar um alarme com a opção de repetir todo dia, poderíamos utilizar a constante `AlarmManager.INTERVAL_DAY`.

```
// Agenda este alarme com a opção de repetir todos os dias no mesmo horário.
AlarmUtil.scheduleRepeat(this, intent, getTime(), AlarmManager.INTERVAL_DAY);
```

26.5 Classe Calendar

Para disparar um alarme, é necessário fornecer o tempo em milissegundos, ou seja, a data e hora. No exemplo que fizemos anteriormente, o objetivo era pegar a data atual e somar cinco segundos, assim o alarme seria agendado para daqui a cinco segundos. Isso foi possível com a ajuda da classe `java.util.Calendar`.

```
Calendar c = Calendar.getInstance();
c.setTimeInMillis(System.currentTimeMillis());
c.add(Calendar.SECOND, 5); // 5 segundos
long time = c.getTimeInMillis();
```

Meu objetivo aqui não é dar uma aula de `Calendar`, até porque isso é assunto para um livro sobre a linguagem Java. Mas, apenas para dar uma dica, essa classe contém métodos para controlar facilmente datas e horas, assim como fazer operações como aumentar uma hora, aumentar um dia etc. Outro exemplo interessante é como configurar o `Calendar` com a data de hoje às 9:30 am.

```
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.set(Calendar.HOUR_OF_DAY, 9);
calendar.set(Calendar.MINUTE, 30);
long time = c.getTimeInMillis();
```

Se você quiser que esta data seja criada apenas amanhã, basta somar um dia ao criar no `Calendar`.

```
Calendar calendar = Calendar.getInstance();
calendar.setTimeInMillis(System.currentTimeMillis());
calendar.set(Calendar.HOUR_OF_DAY, 9);
calendar.set(Calendar.MINUTE, 30);
calendar.add(Calendar.DAY_OF_MONTH, 1);
long time = c.getTimeInMillis();
```

Como eu disse, meu objetivo não é dar uma aula sobre `Calendar`; portanto, caso você não conheça essa classe, recomendo que a estude.

26.6 Quando utilizar ou não um alarme

Para finalizar este capítulo, veremos uma lista de quando é necessário utilizar um alarme e quando seu uso não é recomendado.

É recomendável utilizar um alarme para:

1. Agendar uma aplicação para ser executada em determinado momento no futuro, ou seja, agendar o disparo de uma intent.
2. Um serviço não deve ficar executando eternamente para não utilizar recursos e memória demais. É aconselhável utilizar um alarme para agendar o serviço na data e na hora desejadas e se necessário repetir o alarme para acordar o serviço.

Não é recomendável utilizar um alarme para fazer uma thread dormir e acordá-la em determinado momento. Para isso, é necessário utilizar a classe `Handler`, a qual já estudamos. Caso você quera executar operações de tempos em tempos dentro de uma activity, a classe `Handler` também deve ser utilizada, pois ela contém os métodos `postDelayed(Runnable, long)` e `postAtTime(Runnable, long)`. Lembre-se de que o `Handler` fica atrelado ao ciclo de vida de uma activity, por isso utilizá-lo nesses casos é recomendado.

O alarme deve ser utilizado justamente quando a aplicação não está executando, ou provavelmente não está, pois neste caso o `Handler` não dará conta do recado.

Por último, lembre-se: a principal vantagem de um alarme é disparar uma intent na data e na hora desejadas. Um broadcast receiver pode ser utilizado para interceptar a mensagem e acordar a aplicação, mesmo com ela fechada. Ao interceptar uma mensagem com a aplicação fechada, lembre-se de que o recomendado é mostrar uma notificação para o usuário.

26.7 Links úteis

Neste capítulo, aprendemos a agendar alarmes. Para continuar seus estudos, separei um link da documentação oficial.

- **Android Training – Scheduling Repeating Alarms**

<https://developer.android.com/training/scheduling/alarms.html>



CAPÍTULO 27

Service e JobInfo

A classe `Service` é utilizada para executar um serviço em segundo plano, geralmente vinculado a algum processo que deve executar por tempo indeterminado, e tem um alto consumo de recursos, memória e CPU.

Um `Service` também pode ser utilizado para criar aplicações que são executadas em segundo plano, sem a necessidade de exibir uma interface ao usuário.

Neste capítulo, aprenderemos a criar serviços que ficam executando em segundo plano e entenderemos como funciona o seu ciclo de vida.

27.1 Introdução

A classe `android.app.Service` é utilizada no Android para executar um processamento em segundo plano por tempo indeterminado, chamado popularmente de serviço. Um serviço geralmente faz um alto consumo de recursos, memória e CPU. Não precisa interagir com o usuário e consequentemente não precisa de interface gráfica.

Geralmente um serviço com a classe `Service` é iniciado a partir de um `BroadcastReceiver`, o qual precisa executar rapidamente e retornar do método `onReceiveIntent(context, intent)` o mais breve possível. Por isso, as classes `BroadcastReceiver` e `Service` formam uma dupla comum no Android, pois o receiver intercepta a mensagem enviada pela intent, e se necessário um serviço é iniciado para executar o processo, uma vez que o receiver precisa terminar em dez segundos.

A classe `Service` faz parte do ciclo de vida dos processos controlados pelo sistema operacional, e seu processo não será finalizado enquanto estiver em execução, a não ser que as condições de memória do celular estejam realmente baixas, situação na qual o Android pode decidir encerrar alguns processos para liberar memória e recursos.

O interessante é que, mesmo que um Service seja encerrado devido à falta de memória, o Android posteriormente tentará reiniciá-lo para que o processamento continue assim que as condições de memória e os recursos utilizados estejam normais. Nesse caso, o serviço deve ser codificado de uma forma que seja possível recuperar o estado em que o processamento foi encerrado anteriormente.

27.2 Exemplos de serviços

Geralmente, ao estudar serviços pela primeira vez, é comum confundir o conceito de threads e serviços, pois ambos executam de forma assíncrona e teoricamente em segundo plano.

Primeiramente, não confunda o termo “serviço” no Android com “web service”. Web service é um tipo de serviço que executa lá no servidor web e tem como objetivo fornecer conteúdo para que os aplicativos consultem informações. No Android, se você ouvir o termo “serviço”, provavelmente é um atalho para referenciar a classe `android.app.Service`.

Também não confunda o conceito de threads com um serviço em segundo plano. Uma thread deve ser utilizada sempre que o objetivo for desvincular um processamento da thread principal. É o caso de quando uma activity precisa buscar informações em um web service ou banco de dados porque, a fim de não travar a interface, uma nova thread deve ser iniciada. Um serviço criado pela classe `Service` tem um propósito diferente e pode até executar sem uma activity estar aberta.

A seguir, podemos ver uma lista de possíveis situações nas quais um serviço poderia ser utilizado, para você entender bem o conceito.

- Quando você instala um aplicativo pelo Google Play, é utilizado um serviço. Veja que enquanto o download está sendo feito é mostrada uma notificação para o usuário acompanhar o status, e o usuário está livre para acessar qualquer outro aplicativo. Claramente podemos ver que um serviço fica executando sem a necessidade de a aplicação ou activity estar aberta.
- Você criou um jogo, mas, para diminuir o tamanho do download inicial do aplicativo, todos os artefatos do jogo como imagens, mapas, cenários, sons etc. ficaram no servidor. Na primeira vez que o usuário instalar o jogo, ele terá de fazer o download de todos esses arquivos. Como esse processo pode demorar, o recomendado é iniciar um serviço. Durante o download, o usuário pode continuar fazendo outra coisa, como navegar na internet.

- Você recebeu uma mensagem de push do servidor informando que existem atualizações importantes que seu aplicativo tem de fazer. Caso você precise consultar um web service para isso, pode utilizar um serviço. Tudo deve ocorrer sem atrapalhar a atividade do usuário.
- Por último, vou citar um aplicativo de checklist de carros, o qual desenvolve para uma empresa de auditoria. Basicamente, o aplicativo ajudava a fazer uma inspeção no carro, e, caso alguma avaria (risco, amassado, batida) fosse encontrada, ela era registrada no aplicativo e várias fotos eram tiradas. O operador do aplicativo que fazia as inspeções podia até trabalhar de forma offline o dia inteiro. No final do dia, quando uma conexão Wi-Fi estivesse disponível, o aplicativo fazia o sincronismo com a internet. Isso era feito com um serviço em segundo plano, sem atrapalhar a atividade do usuário. No final do sincronismo das informações, o aplicativo mostrava uma notificação ao usuário, informando que todas as inspeções tinham sido coletadas corretamente.

27.3 Como iniciar e parar um serviço

Depois da teoria inicial, vamos aprender como iniciar um serviço. Primeiro, é necessário criar uma subclasse de `android.app.Service` e configurar o arquivo *AndroidManifest.xml*, como todas as outras classes do Android, adicionando uma simples tag `<service>`, como mostrado a seguir:

```
<service android:name=".HelloService" />
```

E para iniciar o serviço basta chamar o método `startService(intent)`:

```
startService(new Intent(this, HelloService.class));
```

Caso prefira disparar o serviço com uma ação, utilize a tag `<intent-filter>` para configurar a ação que deve disparar o serviço. O atributo `android:exported="false"` indica que esse serviço é privado da aplicação, de forma que outras aplicações não podem utilizar essa intent para iniciá-lo.

```
<service android:name=".HelloService" android:exported="false">
  <intent-filter>
    <action android:name="ACAO_PARA_INICIAR_O_SERVICE" />
    <category android:name="android.intent.category.DEFAULT" />
  </intent-filter>
</service>
```

Neste caso, poderíamos iniciar o serviço com o seguinte código:

```
startService(new Intent("ACAO_PARA_INICIAR_O_SERVICE"));
```

Eu particularmente prefiro iniciar o serviço utilizando a primeira opção, ou seja, utilizar uma intent implícita com o nome da classe que quero executar.

O método `startService(intent)` inicia um serviço que fica executando por tempo indeterminado. O interessante do serviço é que ele vai continuar executando mesmo se o usuário sair da aplicação. Ou seja, o serviço é independente da `activity`.

Depois de iniciar o serviço, existem duas maneiras de pará-lo. A primeira, é chamar o método `stopService(intent)` com a mesma intent que foi utilizada para iniciá-lo. A segunda é o próprio serviço se autoencerrar com o método `stopSelf()`.

Para exemplificar quando parar o serviço com um método ou outro, vou explicar utilizando o aplicativo do Google Play como exemplo. Quando o usuário solicita a instalação ou atualização de um aplicativo, o Google Play inicia um serviço. No final do download, o próprio Google Play pode terminar esse serviço, chamando o método `stopSelf()`. Porém, durante o download, o usuário também pode decidir entrar no Google Play e interromper a instalação, talvez porque ele pretende desligar o Wi-Fi ou o celular está ficando sem bateria. Como isso é uma ação iniciada pelo usuário, neste caso utiliza-se o método `stopService(intent)` para solicitar ao Android que encerre o serviço.

27.4 Exemplo prático

Vamos colocar a mão na massa e criar o projeto **HelloService** no Android Studio. O exemplo que vamos criar é um clássico, o qual vai executar um loop com um contador de 0 até 10 imprimindo as mensagens no LogCat. O objetivo é apenas simular que algum processamento está sendo realizado.

Para criar um serviço, é preciso criar uma subclasse de `android.app.Service` e implementar obrigatoriamente o método `IBinder onBind(intent)`. Opcionalmente, também podem ser implementados os métodos `onCreate()`, `onStartCommand(intent, flags, startId)` e `onDestroy()`, referentes ao ciclo de vida da classe `Service`.

A seguir, podemos visualizar a classe de nosso primeiro serviço:



HelloService.java

```
import livroandroid.lib.utils.NotificationUtil;

public class HelloService extends Service {
    private static final int MAX = 10;
    private static final String TAG = "livro";
    protected int count;
    private boolean running;
```

```

@Override
public IBinder onBind(Intent i) {
    // Por enquanto vamos deixar null. Depois vamos estudar isso.
    return null;
}
@Override
public void onCreate() {
    Log.d(TAG, "HelloService.onCreate() - Service criado");
}
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Log.d(TAG, "HelloService.onStartCommand() - Service iniciado: " + startId);
    count = 0;
    // Método chamado depois do onCreate(), logo depois que o serviço é iniciado
    // O parâmetro 'startId' representa o identificador deste serviço
    running = true;
    // Delega para uma thread
    new WorkerThread().start();
    // Chama a implementação da classe mãe
    return super.onStartCommand(intent, flags, startId);
}
// Thread que faz o trabalho pesado
class WorkerThread extends Thread {
    public void run() {
        try {
            while (running && count < MAX) {
                // Simula algum processamento
                Thread.sleep(1000);
                Log.d(TAG, "HelloService executando... " + count);
                count++;
            }
            Log.d(TAG, "HelloService fim.");
        } catch (InterruptedException e) {
            Log.e(TAG, e.getMessage(), e);
        } finally {
            // Autoencerra o serviço se o contador chegou a 10
            stopSelf();
            // Cria uma notificação para avisar ao usuário que terminou
            Context context = HelloService.this;
            Intent intent = new Intent(context, MainActivity.class);
            NotificationUtil.create(context, 1, intent, R.mipmap.ic_launcher,
                "HelloService", "Fim do serviço.");
        }
    }
}

```

```

    }
}
@Override
public void onDestroy() {
    // Ao encerrar o serviço, altere o flag para a thread parar (isso é importante
    // para encerrar a thread caso alguém tenha chamado o stopService(intent)
    running = false;
    Log.d(TAG, "HelloService.onDestroy() - Service destruído");
}
}

```

Para o código compilar, importe a classe `livroandroid.lib.utils.NotificationUtil` da biblioteca `android-utils`. Feito isso, configure o serviço no arquivo de manifesto.

AndroidManifest.xml

```

<application . . >
    <activity android:name=".MainActivity" ... />
    <service android:name=".HelloService" />
</application>

```

Observe que no código da classe `HelloService` não tem nada de muito complicado. Basta implementar os métodos do ciclo de vida e o método `onBind(intent)`. O importante é entender que um service executa na thread principal da aplicação, podendo prejudicar seu desempenho. Portanto, é recomendado utilizar uma thread ou um `AsyncTask` dentro do service, para executar o processamento pesado. No código eu chamei essa thread de `WorkerThread`. Apenas para simular um processamento demorado, a thread está fazendo um loop até 10 e chamando o método `Thread.sleep(1000)` para dormir por um segundo a fim de demorar um pouco para terminar o serviço. Quando o valor do contador chegar a 10, o loop da thread termina, e o método `stopSelf()` é chamado, o qual encerra o ciclo de vida do serviço, fazendo com que o Android chame o método `onDestroy()`, encerrando o processo para liberar memória e recursos utilizados. Outra forma de parar um serviço é simplesmente chamando o método `stopService(intent)`.

Nota: caso o serviço que estava executando em segundo plano precise informar ao usuário que a execução terminou, utilize uma notificação. Um exemplo disso é quando você instala um aplicativo pelo Google Play. Nesse caso, um serviço é iniciado para baixar o aplicativo, e quando a instalação termina uma notificação é utilizada para avisar ao usuário.

Para mostrar como iniciar e parar o serviço, vamos criar uma activity com os botões **Start** e **Stop** no layout.



/res/layout/activity_main.xml

```
<LinearLayout android:orientation="vertical" ... >
    <TextView
        android:text="Exemplo de serviço, verifique os logs no LogCat."
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <Button
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="Start" android:onClick="onClickStart" />
    <Button
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="Stop" android:onClick="onClickStop" />
</LinearLayout>
```

No código da activity, implemente os métodos para tratar os eventos dos botões a fim de iniciar e parar o serviço. Observe que a mesma intent usada para iniciar o serviço é utilizada para pará-lo.



MainActivity.java

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void onClickStart(View view) {
        startService(new Intent(this, HelloService.class));
    }
    public void onClickStop(View view) {
        stopService(new Intent(this, HelloService.class));
    }
}
```

Pronto! Feito isso, execute o projeto e clique no botão **Start** para iniciar o serviço. O resultado da execução você pode conferir no LogCat, que deve mostrar as seguintes mensagens, que mostram o serviço executando e incrementando o contador de 0 a 10. Você pode inclusive sair da aplicação que o serviço vai continuar executando, pois esse é o seu propósito.

```
D/livro(2606): HelloService.onCreate() - Service criado
D/livro(2606): HelloService.onStartCommand() - Service iniciado: 1
D/livro(2606): HelloService executando... 0
D/livro(2606): HelloService executando... 1
D/livro(2606): HelloService executando... 2
D/livro(2606): HelloService executando... 3
D/livro(2606): HelloService executando... 4
D/livro(2606): HelloService executando... 5
D/livro(2606): HelloService executando... 6
D/livro(2606): HelloService executando... 7
D/livro(2606): HelloService executando... 8
D/livro(2606): HelloService executando... 9
D/livro(2606): HelloService fim thread: 10
D/livro(2606): HelloService.onDestroy() - Service destruído
```

Observe que o próprio serviço encerrou seu processo com a chamada do método `stopSelf()`. Neste exemplo, caso o botão **Stop** seja chamado antes do fim, o serviço será finalizado antes de o contador chegar a 10.

Embora para acompanhar o resultado deste exemplo você deva olhar no LogCat, a figura 27.1 mostra a notificação que é criada quando o serviço terminar por conta própria, ou seja, o contador chegar a 10. Notificações são muito utilizadas nesses casos para avisar ao usuário que um serviço terminou, se for algo que ele estiver esperando, como uma atualização de dados do aplicativo.

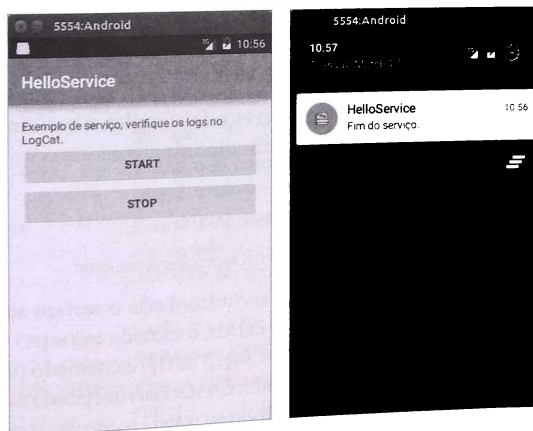


Figura 27.1 – Fim do serviço mostrando uma notificação.

Isso é tudo. Para criar um serviço, basta criar uma subclasse de `android.app.Service` e declarar a classe no arquivo *AndroidManifest.xml* com a tag `<service>`. Depois, o sistema operacional do Android se encarrega do resto, executando a classe em segundo plano até que ela mesmo encerre o processamento com a chamada do método `stopSelf()` ou alguém chame o método `stopService(intent)`.

Nota: lembre-se de que a classe `Service` executa na thread principal da aplicação. Portanto, para executar o processamento, é recomendado iniciar uma thread dentro do serviço. Caso contrário, o serviço poderá prejudicar o desempenho do aplicativo que está interagindo com o usuário.

27.5 Deixar o serviço executando depois de sair de uma tela

O que acontece com o exemplo anterior, se, logo depois de iniciar o serviço, o usuário sair da tela? A resposta é que a `activity` será destruída, mas o serviço não. Ele continuará executando em segundo plano automaticamente. Para encerrar o serviço, basta iniciar a `activity` novamente e pressionar o botão **Stop** ou esperar que o serviço termine por conta própria.

Enfim, este breve tópico foi apenas para lembrá-lo de fazer o teste de fechar a `activity` e deixar o serviço executando, pois essa é a verdadeira essência de um serviço.

27.6 Entendendo o ciclo de vida de um serviço

O primeiro exemplo que demos de um serviço foi simples, e basicamente é isso o que você precisa fazer. Neste tópico, vamos discutir alguns aspectos mais avançados que é bom você dominar.

A figura 27.2 mostra o ciclo de vida da classe `Service`.

Fonte: <http://developer.android.com/guide/components/services.html>

Ao chamar o método `startService(intent)`, o Android cria o serviço apenas se ele não estiver em execução. Se o serviço não existir, o método `onCreate()` é chamado. Logo depois, o método `onStartCommand(intent, flags, startId)` é chamado para iniciar a execução do serviço. É possível chamar o método `startService(intent)` várias vezes, e se isso acontecer o Android chamará o método `onCreate()` somente da primeira vez. Nas outras vezes, apenas o método `onStartCommand(intent, flags, startId)` é chamado, informando o id da execução desse serviço, que vamos verificar adiante. Quando o serviço for finalizado, o método `onDestroy()` é chamado.

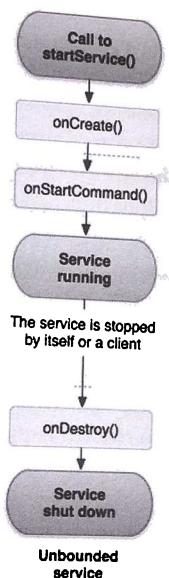


Figura 27.2 – Ciclo de vida da classe Service.

Observe que o método `onStartCommand(intent, flags, startId)` retorna um número inteiro, o qual deve ser uma das constantes a seguir, que definem o comportamento que ocorre caso o Android encerre o processo do serviço por motivos de falta de recursos e memória, entre outros.

Constante	Descrição
START_NOT_STICKY	Indica que, caso o sistema encerre o service, ele não será recriado, a não ser que ainda existam intents a serem entregues.
START_STICKY	Este é o retorno padrão caso o <code>super.onStartCommand(...)</code> da classe mãe seja chamado. Indica que, caso o sistema encerre o service, ele será recriado, e o método <code>onStartCommand(...)</code> será chamado novamente. Mas, neste caso, a intent recebida como parâmetro será nula. Se você não precisar dessa intent, essa constante deverá ser utilizada.
START_REDELIVER_INTENT	Idem à anterior, mas a mesma intent que iniciou o service é entregue novamente. Se o service utiliza a intent para diferenciar o conteúdo, como cada thread pode fazer o upload de diferentes arquivos, utilize essa constante para receber a intent novamente, caso o service seja interrompido.

Nota: não importa quantas vezes o método `startService(intent)` é chamado: uma única chamada ao método `stopService(intent)` finalizará o serviço.

Conhecer o ciclo de vida da classe `Service` é importante, pois a maneira como você vai gerenciar as threads dentro do serviço vai depender do que você precisa fazer. Se você deseja que exista apenas uma thread, poderá criá-la no método `onCreate()`, de forma que novas chamadas não afetem a thread que já está executando.

Você também pode criar uma nova thread a cada chamada do método `onStartCommand(intent, flags, startId)`, mas deve se preocupar com o fato de o código ser multi-threading. Nesse caso, o parâmetro `startId` pode identificar a thread.

Esse parâmetro `startId` deve ser armazenado para encerrar o serviço com o método `stopSelf(startId)`. Veja que esse método recebe o id da thread que precisa ser interrompida, diferentemente do `stopSelf()`, que é utilizado para destruir todos os serviços.

Dica: no projeto de exemplo deste capítulo, você vai encontrar a classe `HelloService_WorkerThread`, que demonstra como criar uma thread a cada chamada do método `onStartCommand(intent, flags, startId)`. Assim, cada vez que o método `startService(intent)` for chamado, uma thread separada irá executar.

27.7 A classe `IntentService`

Até o momento, vimos como criar um service com a classe `android.app.Service`. Agora vamos estudar a classe `android.app.IntentService`, que facilita um pouco esse trabalho.

Conforme estudamos anteriormente, um service executa na mesma thread da aplicação, portanto é recomendado iniciar uma thread separada para cuidar do processamento, a fim de não prejudicar o desempenho da aplicação. Nós já estudamos o ciclo de vida da classe `Service` e vimos que a cada chamada do método `startService(intent)` é chamado o método `onStartCommand(intent, flags, startId)`, o qual é utilizado para fazer uma programação multi-threading, se necessário.

Porém, na maioria das vezes isso não é preciso, nem mesmo recomendado, pois você precisa evitar problemas de concorrência. Por isso foi criada a classe `IntentService`, que é filha de `Service` e implementa todos os seus métodos. Para utilizar a classe `IntentService`, você só precisa implementar o método `onHandleIntent(Intent intent)`, e a `intent` é utilizada para ler os parâmetros enviados ao iniciar o serviço.

A vantagem de utilizar a classe `IntentService` é que ela já cria uma thread dentro dela, inclusive com uma fila de execução. Sendo assim, cada chamada ao método `startService(Intent)` entra nessa fila, que é processado pela `IntentService`. Portanto, é garantido para você que sempre o método `onHandleIntent(Intent intent)` será chamado sequencialmente, sem nenhum problema de concorrência. Enfim, a classe `IntentService` faz o que provavelmente você teria de fazer se ela não existisse.

Outra vantagem da classe `IntentService` é que ela chama automaticamente o método `stopSelf()` ou `stopSelf(startId)` no final da execução; portanto, como eu disse antes, você só precisa implementar o método `onHandleIntent(Intent intent)`.

O código de exemplo a seguir mostra como implementar um serviço com a classe `IntentService`. Estamos utilizando o mesmo exemplo do contador de 0 a 10. Note que não é preciso criar uma thread nem chamar o `stopSelf()`, pois o `IntentService` já faz tudo isso internamente.

HelloIntentService.java

```
public class HelloIntentService extends IntentService {
    public HelloIntentService() {
        super("NomeDaThreadAqui");
    }
    private static final int MAX = 10;
    private static final String TAG = "livro";
    private boolean running;
    @Override
    protected void onHandleIntent(Intent intent) {
        running = true;
        // Este método executa em uma thread
        // Quando ele terminar, o método stopSelf() será chamado automaticamente
        int count = 0;
        while (running && count < MAX) {
            fazAlgumaCoisa();
            Log.d(TAG, "ExemploServico executando... " + count);
            count++;
        }
        Log.d(TAG, "ExemploServico fim.");
    }
    private void fazAlgumaCoisa() {
        try {
            // Simula algum processamento
            Thread.sleep(1000);
        } catch (InterruptedException e) {
```

```

        Log.e(TAG,e.getMessage(),e);
    }
}
@Override
public void onDestroy() {
    super.onDestroy();
    // Ao encerrar o serviço, altera o flag para a thread parar
    running = false;
    Log.d(TAG, "ExemploServico.onDestroy()");
}
}
}

```

Para testar esse serviço, basta executá-lo, e o resultado será o mesmo do exemplo anterior.

```
startService(new Intent(this, HelloIntentService.class));
```

27.8 Criando um player mp3

Serviços geralmente são criados para executar algum processamento pesado ou demorado, como o sincronismo com algum web service.

Mas às vezes o que queremos executar em segundo plano não é necessariamente pesado, mas precisa ser feito apenas pelo fato de que o processo precisa continuar executando. Esse é o caso de um player mp3, pois depois que o usuário escolhe a música e clica no botão **Play** ele provavelmente vai sair do aplicativo para fazer outra coisa. Portanto, a música deve continuar executando.

No capítulo 21, sobre multimídia, criamos um exemplo simples de um player mp3 que tinha os botões **Play**, **Pause** e **Stop**. Porém, ao sair da activity, nós paramos a música, pois é assim que funciona o ciclo de vida da activity.

Neste próximo exemplo, vamos aprimorar o player mp3 que fizemos no capítulo 21 e vamos tocar a música dentro de um serviço. Mas para fazermos isso teremos de praticar alguns conceitos um pouco mais avançados. Não se preocupe, pois vamos fazer tudo passo a passo, e o resultado será muito legal.

Para começar, copie o projeto **PlayerMp3** e renomeie para **PlayerMp3Service**. Esse projeto já deve estar funcionando e tocando um mp3, mas lembre que, do jeito que fizemos o código, o arquivo mp3 precisa existir no SD card.

A primeira classe que vamos implementar é a **Mp3Service**, que é o serviço que vai executar a música em segundo plano.



Mp3Service.java

```

public class Mp3Service extends Service implements InterfaceMp3 {
    private static final String TAG = "livro";
    private PlayerMp3 player = new PlayerMp3();
    private String mp3;
    // Classe filha de Binder para retornar no onBind(intent)
    public class Mp3ServiceBinder extends Binder {
        // Converte para InterfaceMp3
        public InterfaceMp3 getInterface() {
            // Retorna a interface para controlar o Service
            return Mp3Service.this;
        }
    }
    @Override
    public IBinder onBind(Intent intent) {
        // retorna a classe ConexaoInterfaceMp3 para a activity utilizar
        Log.d(TAG, "Mp3Service onBind(). Aqui retorna o IBinder.");
        return new Mp3ServiceBinder();
    }
    @Override
    public void onDestroy() {
        // Fim do serviço
        Log.d(TAG, "Mp3Service onDestroy().");
        // Para a música
        stop();
    }
    // Método da interface InterfaceMp3
    public void start(String mp3) {
        this.mp3 = mp3;
        try {
            player.start(mp3);
        } catch (Exception e) {
            Log.e(TAG, e.getMessage(), e);
        }
    }
    // Método da interface InterfaceMp3
    public void pause() {
        player.pause();
    }
    // Método da interface InterfaceMp3
    public void stop() {
        player.stop();
    }
}

```

```

    }
    // Método da interface InterfaceMp3
    public String getMp3() {
        return mp3;
    }
    // Método da interface InterfaceMp3
    public boolean isPlaying() {
        return player.isPlaying();
    }
}

```

Observe que adicionamos um atributo do tipo `PlayerMp3` à classe do serviço, pois é essa classe que vai tocar a música, e não a `activity`.

Outro detalhe: a classe `Mp3Service` implementa a interface `InterfaceMp3`, portanto crie-a no projeto.



InterfaceMp3.java

```

public interface InterfaceMp3 {
    void play(String mp3); // Inicia a música
    void pause(); // Faz pause da música
    void stop(); // Para a música
    boolean isPlaying(); // Retorna true se está tocando a música
    String getMp3(); // Caminho da música
}

```

O segredo da classe `Mp3Service` é que ela implementou o método `onBind(intent)`, o qual até o momento sempre tinha nos retornado um valor nulo. Isso é necessário para expor a interface de comunicação, que é a `InterfaceMp3`, ao código cliente que está utilizando o serviço, que neste caso é a `activity`. O serviço implementa a interface `InterfaceMp3`, pois ela é chamada de interface de comunicação e será utilizada para se comunicar com o serviço. No método `onBind(intent)` é retornada a classe interna `Mp3ServiceBinder`, a qual permite obter a interface `InterfaceMp3` com o método `getInterface()`. Isso será utilizado pela `activity`, quando ela se conectar ao serviço que está tocando a música.

Dica: o conceito de interface na Orientação a Objetos é bem difundido e ajuda a separar as responsabilidades da aplicação, além de frequentemente ser usado para fazer com que uma camada ou serviço da aplicação se comunique com outra. Assim, a aplicação que vai reproduzir a música não precisa conhecer detalhes de como o código foi implementado; basta chamar os métodos dessa interface.



Na prática, a activity vai se conectar ao serviço, obter uma referência para a interface `InterfaceMp3` e chamar os seus métodos `play(mp3)`, `pause()`, `stop()` etc. Agora vamos partir para o código da activity, que é a parte mais complicada.



MainActivity.java

```
import livroandroid.lib.utils.NotificationUtil;

public class MainActivity extends AppCompatActivity {
    private static final String TAG = "livro";
    // Classe que encapsula o MediaPlayer
    private EditText text;
    private InterfaceMp3 interfaceMp3;
    private ServiceConnection conexao = new ServiceConnection() {
        public void onServiceConnected(ComponentName className, IBinder service) {
            // (*3*)
            // Recupera a interface para interagir com o serviço
            Mp3Service.Mp3ServiceBinder conexao = (Mp3Service.Mp3ServiceBinder) service;
            interfaceMp3 = conexao.getInterface();
            Log.d(TAG, "onServiceConnected, interfaceMp3 conectada: " + interfaceMp3);
        }
        public void onServiceDisconnected(ComponentName className) {
            // (*6*)
            Log.d(TAG, "onServiceDisconnected, liberando recursos.");
            interfaceMp3 = null;
        }
    };
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.activity_main);
        text = (EditText) findViewById(R.id.tArquivo);
        Intent intent = new Intent(this, Mp3Service.class);
        Log.d(TAG, "Iniciando o service");
        // (*1*)
        startService(intent);
        // Faz o bind/ligação
        // (*2*)
        boolean b = bindService(intent, conexao, Context.BIND_AUTO_CREATE);
        Log.d(TAG, "Service conectado: " + b);
    }
    public void onClickPlay(View view) {
        // (*4*)
        if(interfaceMp3 != null) {
```



```

        String mp3 = text.getText().toString();
        Log.d(TAG, "play: " + mp3);
        interfaceMp3.play(mp3);
    }
}

public void onClickPause(View view) {
    // (*4*)
    if(interfaceMp3 != null) {
        Log.d(TAG, "pause");
        interfaceMp3.pause();
    }
}

public void onClickStop(View view) {
    // (*4*)
    if(interfaceMp3 != null) {
        Log.d(TAG, "stop");
        interfaceMp3.stop();
    }
}

@Override
protected void onStop() {
    super.onStop();
    if(interfaceMp3 != null && interfaceMp3.isPlaying()) {
        // (*5*)
        Log.d(TAG, "Activity destruida. A música continua.");
        unbindService(conexao);
        // Cria a notificação para o usuário voltar ao player.
        String mp3 = interfaceMp3.getMp3();
        NotificationUtil.create(this, 1, new Intent(this, MainActivity.class), "MP3 Player", mp3);
    } else {
        // (*7*)
        Log.d(TAG, "Activity destruida. Para o serviço, pois não existe música tocando.");
        unbindService(conexao);
        stopService(new Intent(this, Mp3Service.class));
    }
}
}
}

```

Antes de eu explicar o código, execute o projeto no emulador e clique no botão **Play** para tocar a música. Lembrando que, para funcionar, o arquivo de mp3 precisa existir no SD card do emulador. O resultado pode ser visto na figura 273. Mesmo que o usuário feche a aplicação, a música vai continuar executando. Ao sair da activity, estamos inserindo uma notificação na barra de status, pois isso

é um padrão comum para indicar que existe um serviço executando e o usuário pode clicar na notificação para voltar ao aplicativo.

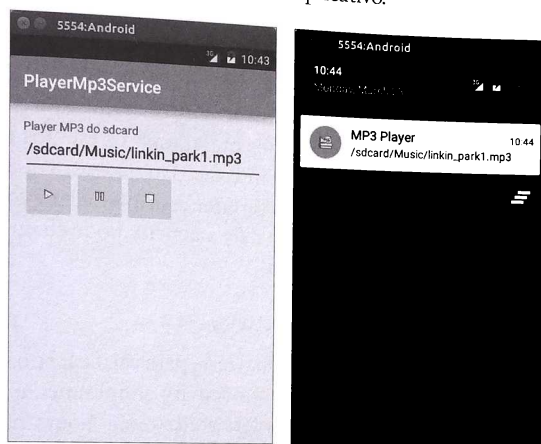


Figura 27.3 – Player mp3.

Agora vamos entender o código da `MainActivity`. Para facilitar a explicação, observe que deixei marcações no código.

(1) Inicia o serviço

O método `startService(intent)` foi chamado logo ao abrir a activity para iniciar o serviço em segundo plano. Esse serviço por enquanto não faz nada, mas permanece vivo aguardando a próxima instrução. Lembre-se de que um serviço faz parte do ciclo de vida da plataforma do Android, e portanto o Android vai deixá-lo executando lá.

(2) Conecta ao serviço

Aqui está o segredo de tudo. O método `bindService(intent, con, flags)` precisa passar nos parâmetros um objeto que implemente a interface `ServiceConnection`, a qual contém métodos de callback para quando o código da activity é conectado ou desconectado do serviço.

(3) Recupera a interface de comunicação

Se a chamada do método `bindService(intent, con, flags)` for bem-sucedida o código irá se conectar ao serviço e o método `onServiceConnected()` será chamado. Nesse momento, a interface de comunicação `InterfaceMp3` é recuperada.

Tudo consiste em programação Java, mas se tiver dificuldades em entender o código não se preocupe, continue lendo e depois tente novamente. Na segunda vez é sempre mais fácil.

(4) Usuário clica nos botões Play, Pause e Stop

Uma vez que o serviço foi iniciado e a activity conectou-se a ele, podemos tocar a música. Portanto, os métodos que tratam os eventos da tela simplesmente chamam os métodos da interface `InterfaceMp3`, a qual foi obtida quando nos conectamos ao serviço. Veja que essa interface de comunicação é o segredo de tudo, pois por meio dela a activity pode chamar métodos que serão executados lá no serviço.

(5) A música fica tocando mesmo se sair da activity

No método `onStop()` da activity é feito o teste para verificar se o serviço está tocando alguma música. Se estiver, a activity simplesmente desconecta do serviço chamando o método `unbindService(conn)`. Como nesse caso o serviço continua executando, uma notificação é criada para mostrar ao usuário qual música está tocando, para permitir que o usuário volte para o aplicativo de forma fácil.

(6) Remove a interface de comunicação

Logo depois de se desconectar do serviço, o método `onServiceDisconnected()` é chamado. Nesse momento, o objeto que contém a interface de comunicação não é mais válido.

(7) Encerrando o serviço caso não esteja tocando nenhuma música

No método `onStop()` da activity, caso a activity não esteja tocando nenhuma música, não tem razão para deixar o serviço executando em segundo plano. Por isso, o método `stopService(intent)` é chamado para encerrá-lo, liberando os recursos e memória.

Esse exemplo é avançado, mas demonstrou como uma activity pode se conectar a um serviço que está executando em segundo plano. Vimos que por meio da interface de comunicação, a qual é exposta pelo serviço, a activity pode invocar métodos lá dentro da classe do serviço.

Entenda que o serviço é iniciado pelo método `startService(intent)` e permanece executando até que o método `stopService(intent)` seja chamado. Sempre que a activity for iniciada, independentemente de o serviço já estar executando ou

não, o método `bindService(intent, con, flags)` é utilizado para se conectar ao serviço. Ao encerrar a `activity`, o método `unbindService(conn)` é chamado para desconectar.

Isso significa que podemos nos conectar e desconectar de um serviço diversas vezes a fim de obter a interface de comunicação e chamar algum método dentro da classe do serviço. E, quando o serviço não for mais necessário, o método `stopService(intent)` é chamado para encerrar todo o processo.

27.9 Método `bindService(intent, con, flags)`

Caso você precise de mais informações sobre o método `bindService(intent, con, flags)` que estudamos no tópico anterior, este tópico visa complementar o assunto.

O método `bindService(intent, con, flags)` pode iniciar um serviço se ele ainda não estiver executando ou simplesmente conectar-se a ele. Ao estabelecer a conexão, é possível recuperar uma referência para a interface de comunicação.

Espere, você prestou atenção na frase? Eu disse que este método pode iniciar um serviço ou se conectar a ele. Isso significa que não necessariamente você precisaria chamar o método `startService(intent)` para iniciar o serviço. Conforme estudamos, o método `startService(intent)` é utilizado para iniciar um serviço, mas também podemos fazer isso usando o método `bindService(intent, con, flags)`.

O método `bindService(intent, con, flags)` pode iniciar um serviço se ele ainda não estiver executando chamando o método `onCreate()` da classe `Service`, mas ele não chama o método `onStartCommand()`. Esse método é usado principalmente para se conectar ao serviço, obtendo uma referência de uma classe ou interface que pode ser utilizada para se comunicar e chamar métodos do serviço. Essa interface é do tipo `android.os.IBinder`, e a aplicação está conectada ao serviço enquanto manter uma referência para ela.

A seguir, podemos ver a descrição dos parâmetros deste método.

Parâmetro	Descrição
<code>intent</code>	Intent para executar o serviço.
<code>conexao</code>	Implementação da interface <code>android.content.ServiceConnection</code> , que define os métodos <code>onServiceConnected(classe, ibinder)</code> e <code>onServiceDisconnected(classe)</code> , os quais são chamados para notificar que uma conexão com o serviço foi realizada ou encerrada, respectivamente.
<code>flags</code>	0 ou <code>BIND_AUTO_CREATE</code> : se for <code>BIND_AUTO_CREATE</code> , o serviço é automaticamente criado, caso seja necessário.

A figura 27.4 mostra o ciclo de vida de um serviço executado com o método `bindService(intent, con, flags)`.

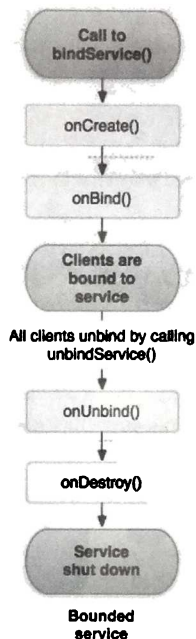


Figura 274 – Ciclo de vida ao chamar o método `bindService()`.

Conforme vimos anteriormente, o método `startService(intent)` tem por finalidade maior iniciar um serviço completamente desvinculado do processo principal de quem o criou. Já o objetivo principal do método `bindService(intent, con, flags)` é estabelecer uma conexão com um serviço já em execução, para literalmente chamar métodos da classe que representa e controla esse serviço.

No caso de o serviço ser iniciado pelo método `bindService(intent, conexao, flags)`, é obrigatório implementar corretamente o método `onBind(intent)`, que precisa retornar uma referência para uma interface de comunicação, a qual deve expor métodos para controlar o estado do serviço ou consultar informações.

Nota: ao criar um serviço com o método `bindService(intent, conexao, flags)`, o processo fica vinculado a quem o criou, que, neste caso, é a `activity`. Isso significa que, se a `activity` for encerrada, o serviço também será. É o contrário do método `startService(intent)`.

27.10 Qual método utilizar para iniciar um serviço?

Agora que vimos as duas maneiras de se conectar a um serviço, é importante entender a diferença em relação ao seu ciclo de vida, dependendo da forma como ele é iniciado.

Entenda que ambos os métodos `startService(intent)` e `bindService(intent,conexao,flags)` podem iniciar o serviço. Com o método `bindService(intent,conexao,flags)`, o processo do serviço está vinculado a quem se conectou a ele, por exemplo, uma `activity`. Então, se a `activity` for encerrada, o serviço também será. Já o método `startService(intent)` deixa o serviço executando por tempo indeterminado, de forma independente da `activity`, até que alguém chame o método `stopService(intent)`.

Com o método `startService(intent)`, é possível deixar o serviço executando em segundo plano mesmo depois de o processo que o criou ser finalizado. Mas somente com o método `bindService(intent,conexao,flags)` é possível conectar-se ao serviço e recuperar a interface de comunicação para interagir com ele.

Então, qual método utilizar para iniciar um serviço? O melhor é utilizar os dois juntos. Por exemplo, é possível iniciar o serviço com o método `startService(intent)` para garantir que o serviço fique executando continuamente e depois chamar o método `bindService(intent,conexao,flags)` para se conectar ao serviço já iniciado e obter a interface de comunicação. Posteriormente, os métodos `unbindService(conexao)` e `stopService(intent)` podem ser chamados para desconectar do serviço ou destruí-lo, respectivamente, conforme a situação desejada. Foi isso que fizemos no exemplo do mp3 player.

27.11 Um serviço em execução contínua não consome muito processamento?

Um fator importante que deve ser considerado ao desenvolver um serviço é que não é recomendado deixá-lo executando continuamente para não consumir muitos recursos.

Imagine que seu serviço precise baixar atualizações de um servidor web, mas antes tem de verificar se existem atualizações disponíveis. Ele poderia ficar dormindo de minuto em minuto para realizar essas verificações, o que poderia consumir muito processamento.

Basicamente existem duas maneiras de resolver essa situação, tudo depende na verdade do que você precisa fazer.

A primeira maneira é utilizar um alarme criado com a classe `android.app.AlarmManager` e agendar uma intent para acordar o serviço na data desejada. Os alarmes poderiam ser utilizados para executar o serviço de hora em hora, ou executá-lo todos os dias à meia-noite, por exemplo. A maneira que gosto de fazer é disparar um alarme que vai acordar um broadcast receiver, que por sua vez apenas inicia o serviço chamando o método `startService(intent)`.

A segunda maneira é o serviço não ficar buscando dados no servidor, na tentativa de verificar se existem atualizações. O melhor seria o servidor enviar uma mensagem de push para o dispositivo, para avisar à aplicação que existem atualizações. Vamos estudar sobre mensagens push no próximo capítulo, mas basicamente o push é uma mensagem enviada do servidor para o dispositivo, a qual utiliza a conexão com a internet como canal de comunicação. No aplicativo, o push é recebido por meio de um broadcast receiver. Neste caso do serviço, quando a aplicação receber a mensagem de push, ela poderia buscar os dados no servidor. Isso economiza recursos e bateria do dispositivo, e é bem mais eficiente do que ficar de tempos em tempos perguntando ao servidor se existem atualizações.

Nota: o objetivo de um serviço é executar algum processamento pesado em segundo plano sem interferir na atividade do usuário. Mas você não deve deixá-lo executando para sempre, pois o serviço pode consumir muitos recursos. Utilize alarmes para agendar a data/hora em que o serviço deve executar, ou utilize mensagens push para acordar o aplicativo.

27.12 JobInfo – a nova API do Lollipop

Para fecharmos este capítulo com chave de ouro, vamos estudar uma das APIs mais interessantes que foi criada no Android 5.0 Lollipop, que é a API de Jobs. Um job é uma tarefa que pode ser agendada para executar em determinado momento, desde que ele satisfaça os critérios estipulados como gatilho para essa execução.

Podemos dizer que um job é uma mistura de um alarme e de um serviço que acabamos de estudar. Com a API de alarmes, conseguimos disparar uma intent na hora desejada e até programá-la para repetir de uma em uma hora, ou repetir todos os dias. E o serviço (classe `Service`) é especializado em executar uma tarefa em segundo plano.

O job meio que mistura essas duas APIs. O job é um tipo de serviço que pode executar em determinados momentos, conforme você tenha feito o agendamento, de maneira parecida com a que agendamos um alarme.

A vantagem do job é que o gatilho para ele executar pode ser:

- Existe conexão Wi-Fi.
- O dispositivo foi conectado na USB.
- O dispositivo entrou em modo bloqueio.
- O dispositivo está carregando a bateria.

Você talvez não esteja interessado na hora que o job vai executar, mas sim nas condições/critérios sob as quais ele irá executar. Sendo assim, podemos executar o job sempre que o dispositivo estiver plugado e com uma conexão Wi-Fi disponível. Esta API é de grande ajuda em muitos casos, mas é compatível com Android 5.0 ou superior. Na época em que este livro estava sendo escrito, não existia nenhuma biblioteca de compatibilidade, porém quem sabe o Google logo não construirá uma?

Para agendar um job, basta criar um objeto do tipo `JobInfo` utilizando o `JobInfo.Builder`. Feito isso, basta obter uma referência da classe `JobScheduler` e criar o job, representado pela classe `JobInfo`. Para brincarmos com a API de Jobs, crie um projeto chamado **HelloJobInfo**. Feito isso, crie a classe `JobUtil`, que mostra como agendar um job para executar assim que o dispositivo estiver conectado ao Wi-Fi e estiver carregando.



`JobUtil.java`

```
@TargetApi(Build.VERSION_CODES.LOLLIPOP)
public class JobUtil {
    public static void schedule(Context context, Class<?> cls, int id) {
        // JobService que vai executar
        ComponentName mServiceComponent = new ComponentName(context, cls);
        JobInfo.Builder builder = new JobInfo.Builder(id, mServiceComponent);
        // Wi-Fi
        builder.setRequiredNetworkType(JobInfo.NETWORK_TYPE_UNMETERED);
        // Carregando
        builder.setRequiresCharging(true);
        // Agenda o job
        JobScheduler jobScheduler =
            (JobScheduler) context.getSystemService(Context.JOB_SCHEDULER_SERVICE);
        JobInfo job = builder.build();
        jobScheduler.schedule(job);
    }
}
```



```

    public static void cancel(Context context, int id) {
        JobScheduler jobScheduler = . . .;
        jobScheduler.cancel(id);
    }
    public static void cancelAll(Context context) {
        JobScheduler jobScheduler = . . .;
        jobScheduler.cancelAll();
    }
}

```

Com essa classe utilitária em mãos, vamos criar o layout e o código da activity. Teremos apenas um botão para agendar o job e outro para cancelá-lo. Deixarei o layout com você. Basta criar dois botões e chamar os métodos `onClickAgendar(view)` e `onClickCancelar(view)`, conforme demonstrado a seguir.



MainActivity.java

```

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void onClickAgendar(View view) {
        int id = 1;
        JobUtil.schedule(this,HelloJobService.class, id);
        Toast.makeText(this,"Job agendado",Toast.LENGTH_SHORT).show();
    }
    public void onClickCancelar(View view) {
        JobUtil.cancel(this,1);
    }
}

```

Esse código vai agendar o job da classe `HelloJobService` para executar assim que o dispositivo estiver carregando a bateria e com uma conexão Wi-Fi disponível. A classe `HelloJobService` deve ser filha de `JobService` e deve implementar o método `onStartJob()` responsável pela lógica do job. A classe `JobService` por sua vez é filha direta de `Service`, portanto tudo o que estudamos sobre serviços continua se aplicando. No método `onStartJob()`, você deve iniciar uma thread para desvincular a tarefa do job da thread principal da aplicação.

A seguir, podemos visualizar o código-fonte da classe `HelloJobService`.



HelloJobService.java

```

public class HelloJobService extends JobService {
    private static final String TAG = "livroandroid";
    @Override
    public void onCreate() {
        super.onCreate();
        Log.d(TAG, "onCreate()");
    }
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.d(TAG, "onStartCommand()");
        return super.onStartCommand(intent, flags, startId);
    }
    @Override
    public boolean onStartJob(JobParameters params) {
        Log.d(TAG, "onStartJob(): " + params.getJobId());
        Intent notIntent = new Intent(this, MainActivity.class);
        String title = "Job";
        String contentTitle = "Hello Job: " + params.getJobId();
        NotificationUtil.create(this, R.mipmap.ic_launcher, notIntent, R.mipmap.ic_launcher,
            title, contentTitle);
        return true;
    }
    @Override
    public boolean onStopJob(JobParameters params) {
        Log.d(TAG, "onStopJob()");
        return true;
    }
    @Override
    public void onDestroy() {
        super.onDestroy();
        Log.d(TAG, "onDestroy()");
    }
}

```

Note que o job contém os mesmos métodos de ciclo de vida da classe Service. Apenas os métodos `onStartJob()` e `onStopJob()` são específicos para informar quando o job vai iniciar e parar. Para finalizar a configuração, basta adicionar a tag `<service>` ao *AndroidManifest.xml* e informar a permissão `BIND_JOB_SERVICE`, necessária para utilizar os jobs.

**AndroidManifest.xml**

```
<application . . . >
    <service android:name=".HelloJobService"
        android:permission="android.permission.BIND_JOB_SERVICE" android:exported="true"/>
</application>
```

Pronto! Isso é tudo. Agora execute o projeto no seu Android e clique no botão para agendar o job. Veja que com o celular plugado na USB e com Wi-Fi o job vai executar. Neste exemplo, o job está somente mostrando uma notificação, apenas para você saber quando ele executou.

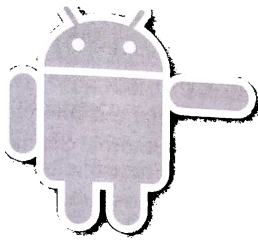
Esta API é nova, mas tem grande potencial. Sua documentação ainda é escassa, e fica até difícil recomendar uma leitura, mas espero que em breve exista algo na documentação oficial. Recomendando também importar no Android Studio pelo menu **Import Sample** o exemplo **JobScheduler**, que com certeza vai ajudá-lo.

27.13 Links úteis

Neste capítulo, aprendemos a executar serviços em segundo plano e fizemos até um exemplo com um player mp3.

Para continuar seus estudos, separei alguns links interessantes.

- **Android API Reference – Service**
<http://developer.android.com/reference/android/app/Service.html>
- **Android API Guides – Service**
<http://developer.android.com/guide/components/services.html>
- **YouTube Google Developers – Using the Android Job Scheduler**
<https://www.youtube.com/watch?v=QdINLG5QrJc>



CAPÍTULO 28

GCM – Google Cloud Messaging

Neste capítulo, vamos estudar o Google Cloud Messaging, um serviço disponibilizado pelo Google para enviar mensagens por push.

28.1 O que é push?

Tradicionalmente, aplicativos utilizam web services para se comunicar com o servidor, e a comunicação é sempre iniciada pelo lado do cliente, ou seja, o dispositivo.

Contudo, esse paradigma atende bem a alguns casos, como por exemplo abrir uma tela para visualizar uma lista de carros ou notícias. Porém, dependendo do caso, esse paradigma de o dispositivo iniciar a conexão pode não ser a melhor solução. Um exemplo clássico que podemos citar é o aplicativo do Gmail. Você já pensou como ele faz para monitorar os novos emails que chegam à sua caixa de entrada?

Para implementar essa verificação, o dispositivo poderia iniciar uma conexão de tempos em tempos repetitivamente com o servidor, para ficar buscando os novos emails. Mas o problema é que você pode receber emails de cinco em cinco minutos ou, às vezes, de cinco em cinco horas; então, de quanto em quanto tempo você deve monitorar o servidor?

Justamente por não saber essa resposta, a solução é inverter os papéis e fazer o servidor acordar o cliente. Isso é possível graças às mensagens de push, que são enviadas do servidor para o aplicativo. Por esse motivo, os dispositivos Android sempre mantêm uma pequena conexão ativa com os servidores do Google, para criar um canal de comunicação que fica dormindo, porém pode receber mensagens a qualquer momento. Esse canal de comunicação é utilizado para que o servidor do Google envie uma mensagem ao dispositivo, para que ele acorde e reaja a determinado

evento. No caso do Gmail, o servidor do Google envia uma mensagem de push informando que existem novos emails disponíveis e que o dispositivo pode, então, iniciar um web service para fazer o sincronismo com o servidor.

Esse processo de o servidor enviar uma mensagem para o celular é conhecido como **push**, e o serviço responsável por implementar essa tarefa no Android chama-se **Google Cloud Messaging** ou, simplesmente, **GCM**.

Nota: o tamanho do conteúdo da mensagem de push que o servidor do Google envia ao dispositivo é pequeno, com limite de 4 bytes. Portanto, ao enviar a mensagem para o dispositivo, utilize um texto pequeno, por exemplo, um simples código, apenas para indicar que existem novos emails disponíveis no servidor. É importante entender que os emails ou qualquer outro tipo de conteúdo não são enviados com a mensagem de push, pois isso excederia facilmente o tamanho da mensagem. Receber uma mensagem de push é como receber um alerta (ping) indicando que existem informações no servidor.

28.2 Como funciona o GCM

Para enviar uma mensagem de push para o dispositivo, é preciso obter o seu código único, chamado de **registration id**. Para isso, o aplicativo deve fazer uma consulta no servidor do GCM e obter o **registration id**, conforme ilustrado na figura 28.1.

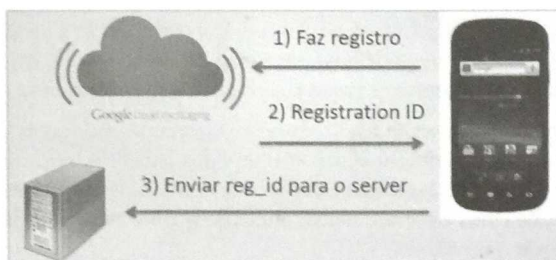


Figura 28.1 – Obtendo um **registration id**.

Depois de obter o **registration id** do dispositivo, é responsabilidade sua salvá-lo em algum lugar. Geralmente os aplicativos enviam o **registration id** para um servidor da empresa, e pode existir uma tabela de usuarios no servidor que vincule determinado usuario do sistema aplicativo ao **registration id**. Dessa forma, quando o servidor precisar se comunicar com o usuario, basta ele consultar o banco de dados e ler o **registration id** do usuario desejado.

Este processo de enviar o **registration id** para um servidor não faremos no livro, pois isso consiste em desenvolver um pequeno sistema, o que está fora do nosso escopo. Mas lembre-se de que no capítulo 17, sobre web services, já mostramos como fazer HTTP Post em web services, portanto você terá todas as condições de fazer algo assim quando precisar.

Para fins de testes, assim que gerarmos o **registration id** no aplicativo, vamos copiá-lo manualmente e inseri-lo no programa que faremos, que por sua vez vai enviar a mensagem de push.

A figura 28.2 demonstra o fluxo para enviar uma mensagem. Observe que o seu servidor nunca vai se comunicar diretamente com o dispositivo. Na prática, a mensagem é enviada ao servidor do GCM, e este por sua vez entrega a mensagem ao dispositivo. Se existir uma conexão disponível com o dispositivo, a mensagem é enviada quase que instantaneamente; caso contrário, o próprio GCM aguarda para enviar a mensagem quando o dispositivo se conectar.

O disposto receberá a mensagem por meio de um broadcast receiver que deve interceptar toda as intents da ação `com.google.android.c2dm.intent.RECEIVE`.

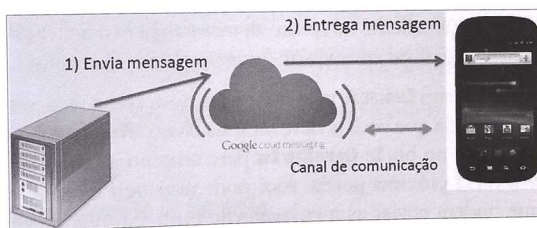


Figura 28.2 – Enviando uma mensagem de push.

28.3 Gerando a chave de acesso do GCM

Para utilizar o GCM, primeiramente é necessário habilitar o serviço na página do Google Developers Console.

<https://console.developers.google.com/>

Como já estudamos a página do console no capítulo 22, sobre mapas, você já sabe utilizá-lo. Se você seguiu minha orientação, deve ter criado um projeto chamado **Livro Android** no qual a API **Google Maps Android API v2** está habilitada.

Desta vez, repita o procedimento, entre no menu **APIs & auth > APIs** e habilite a API **Google Cloud Messaging for Android**, conforme a figura 28.3.

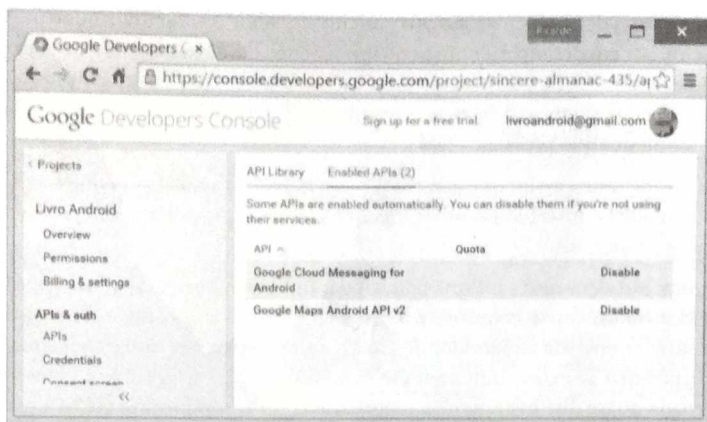


Figura 28.3 – Habilitando o GCM.

E da mesma forma que geramos uma chave para a aplicação dos mapas, desta vez vamos gerar uma chave para o serviço do GCM. Contudo, essa chave não será gerada para a aplicação cliente no Android, mas sim para a aplicação que será o servidor do push, a qual vai enviar as mensagens para o dispositivo.

Portanto, entre no menu **Credentials** para criar uma nova chave de acesso, chamada de **API Key**. Lembre-se de que você já deve ter uma chave **API Key** feita no capítulo 22, sobre mapas. Clique no botão **Create New Key** para criar uma nova chave e escolha a opção **Server Key**. Na próxima janela, você pode restringir os endereços IPs dos servidores que podem enviar as mensagens de push. No nosso exemplo, deixe este campo de texto vazio e clique no botão **Create**.

A figura 28.4 mostra a chave criada. Copie o código do **API Key** antes de prosseguir, pois vamos utilizá-lo posteriormente no servidor para enviar a mensagem de push.

Key for server applications	
API key	AIzaSyCMCQxjMeoAmI8jIiDH4388EJI4qJFZeU
IPs	Any IP allowed
Activation date	Mar 26, 2013, 11:50:00 AM
Activated by	livroandroid@gmail.com (you)
Edit allowed IPs	Regenerate key
	Delete

Figura 28.4 – Gerando a API Key para o servidor do push.

28.4 Obtendo o Project Number

Ainda na página do Google Developers Console, entre na página > **Overview** do projeto e copie o código do projeto, conhecido como **Project Number**, conforme a figura 28.5.

Esse código é utilizado no aplicativo Android para se registrar no GCM a fim de obter o **registration id** que identifica o dispositivo.

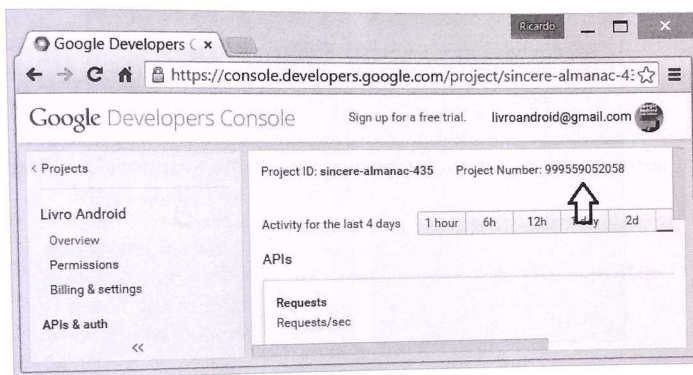


Figura 28.5 – Obtendo o Project Number.

28.5 Executando o projeto de exemplo

O código do projeto Android para fazer o push funcionar é um pouco extenso, portanto recomendo que você teste primeiro o projeto **HelloPush** disponível nos exemplos do livro. Assim você já terá uma boa ideia do que queremos fazer, além de ser legal vermos a mensagem de push chegando ao dispositivo.

Abra o projeto **HelloPush** no Android Studio e altere a constante `PROJECT_NUMBER` da interface `Constants` pelo número do projeto que você criou na sua conta do Google.



Constants.java

```
package br.livro.android.cap28.push;
public interface Constants {
    // Project Number criado na página do Google Developers Console
    String PROJECT_NUMBER = "999559052058";
}
```


Faça essa alteração, execute o projeto em um dispositivo real com Android com o Google Play Services instalado.

Como resultado, é feito o registro no GCM e o **registration id** do dispositivo é impresso na tela (Figura 28.6). Para continuar, copie o **registration id** que aparece nos logs do LogCat, pois vamos utilizá-lo para enviar a mensagem.

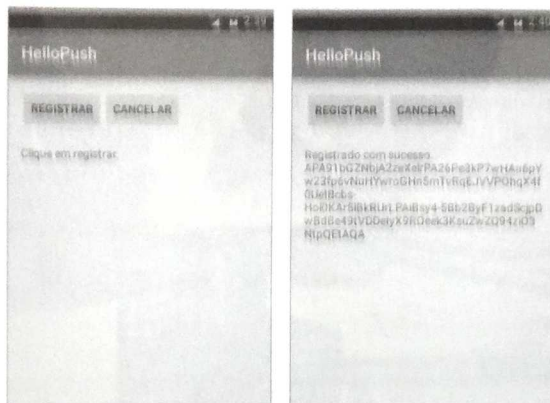


Figura 28.6 - Fazendo o registro no GCM

28.6 Enviando a mensagem de push

Com o **registration id** do dispositivo em mãos, vamos criar um programa em Java para enviar a mensagem.

Para enviar uma mensagem de push ao dispositivo, é necessário fazer um POST no servidor do Google Cloud Messaging, na seguinte URL:

<https://android.googleapis.com/gcm/send>

No POST, precisamos enviar no corpo da requisição (body), o código **registration id** do dispositivo de destino, a mensagem com os parâmetros na URL e, ainda, adicionar ao cabeçalho da requisição HTTP o parâmetro **Authorization**, com valor **key-API key**. Para demonstrar como seria esse código, segue uma implementação utilizando Java.



SendPushMessage.java

```

public class SendPushMessage {
    // Registration id do dispositivo
    private static final String DEVICE_REGISTRATION_ID = "APA91bGZNbjA2zeXelrPA26PeA . . . .";
    // Chave criada no Console. Menu > API Access > (create new server key)
    private static final String API_KEY = "AIzaSyCMCGxjvjeoAmI8jlldH4388EJI4qJFZsU";
    public static void main(String[] args) throws IOException {
        Map<String, String> params = new HashMap<String, String>();
        params.put("msg", "Olá, Leitor");
        String result = post(API_KEY, DEVICE_REGISTRATION_ID, params);
        System.out.println(result);
    }
    // Faz POST no servidor do Google "https://android.googleapis.com/gcm/send"
    public static String post(String apiKey, String deviceRegistrationId, Map<String,
        String> params) throws IOException {
        // Parâmetros necessários para o POST
        StringBuilder postBody = new StringBuilder();
        postBody.append("registration_id").append("=").append(deviceRegistrationId);
        // Cria os parâmetros chave=valor
        Set<String> keys = params.keySet();
        for (String key : keys) {
            String value = params.get(key);
            postBody.append("&").append("data.").append(key).append("=").append(
                URLEncoder.encode(value, "UTF-8"));
        }
        // Cria a mensagem
        byte[] postData = postBody.toString().getBytes("UTF-8");
        // Faz POST
        URL url = new URL("https://android.googleapis.com/gcm/send");
        HttpURLConnection.setDefaultHostnameVerifier(new CustomizedHostnameVerifier());
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setDoOutput(true);
        conn.setUseCaches(false);
        conn.setRequestMethod("POST");
        conn.setRequestProperty("Content-Type",
            "application/x-www-form-urlencoded; charset=UTF-8");
        conn.setRequestProperty("Content-Length", Integer.toString(postData.length));
        conn.setRequestProperty("Authorization", "key=" + apiKey);
        // Lê a resposta
        OutputStream out = conn.getOutputStream();
        out.write(postData);
        out.close();
    }
}

```

```

        int responseCode = conn.getResponseCode();
        if(responseCode == 200) {
            // OK
            String response = conn.getResponseMessage();
            return response;
        } else {
            System.err.println(responseCode + ": " + conn.getResponseMessage());
        }
        return null;
    }
    private static class CustomizedHostnameVerifier implements HostnameVerifier {
        public boolean verify(String hostname, SSLSession session) {
            return true;
        }
    }
}

```

Para testar esse código, crie um projeto Java em alguma IDE, como o Eclipse. Lembrando que uma classe Java pode ser executada no Eclipse pelo menu **Run As > Java Application**.

Mas antes de executar o código altere a constante `DEVICE_REGISTRATION_ID` para o valor que você copiou do seu dispositivo ao se registrar no GCM.

```

// Registration id do dispositivo
private static final String DEVICE_REGISTRATION_ID = "APA91bGZNbjA2zeXelrPA26PeA . . . .";

```

Altere também a constante `API_KEY` com a **API KEY** que você copiou do seu projeto.

```

// Chave criada no Console. Menu > API Access > (create new server key)
private static final String API_KEY = "AIzaSyCMCGxjvjeoAmI8j1LDH4388EJI4qJFZsU";

```

Pronto! Agora execute o código para enviar a mensagem de push para o dispositivo. Ao receber a mensagem, o aplicativo vai mostrar uma notificação, conforme mostra a figura 28.7.

Para finalizar, é importante saber que a requisição HTTP enviada ao servidor do GCM pode retornar os seguintes códigos:

Retorno HTTP	Descrição
200	Indica que a mensagem foi recebida com sucesso, e o servidor do GCM tentará entregar a mensagem.
401	Indica que ocorreu um erro ao autorizar o envio da mensagem. Neste caso, você deve verificar se a chave utilizada está correta.
5xx	Qualquer código de retorno entre 500 a 599 indica que ocorreu um erro no servidor do GCM.

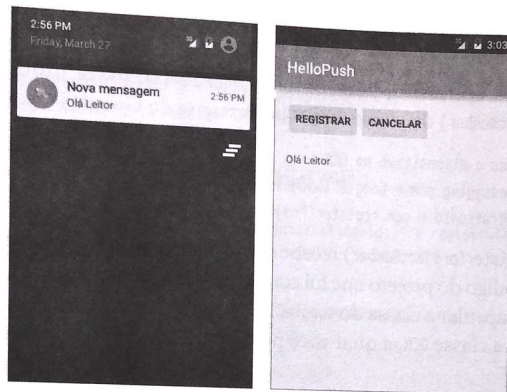


Figura 28.7 – Enviando a mensagem.

Para obter mais detalhes, recomendo ler a documentação oficial.

<https://developer.android.com/google/gcm/index.html>

28.7 Criando o projeto Android passo a passo

Agora que já sabemos como o push funciona, vamos criar o projeto Android passo a passo. Se preferir, apenas abra o projeto de exemplo deste capítulo no Android Studio e acompanhe a explicação.

Se for criar o projeto do zero, crie um projeto com o nome **HelloPush** e o pacote `br.com.livroandroid.hellopush`.

Atenção: eu criei o projeto HelloPush com o pacote `br.com.livroandroid.hellopush`. Caso você crie o seu projeto com outro pacote, tenha muita atenção e leia cuidadosamente as próximas instruções, pois existem configurações feitas no arquivo de manifesto que levam o nome do pacote.

Primeiramente, é necessário declarar a dependência do Google Play Services no arquivo `app/build.gradle`, pois, assim como a API de mapas, o GCM também faz parte do Google Play Services.

 `app/build.gradle`

```
...  
compile 'com.google.android.gms:play-services:7.0.0'
```

28.8 Classe GoogleCloudMessaging

No código do aplicativo, para obter o **registration id**, basta utilizar o método `register(projectNumber)` da classe `GoogleCloudMessaging` do Google Play Services:

```
// Registrando o dispositivo no GCM
GoogleCloudMessaging gcm = GoogleCloudMessaging.getInstance(context);
String registrationId = gcm.register(Project Number Aqui);
```

O método `register(projectNumber)` recebe como parâmetro o **Project Number**, que, lembrando, é o código do projeto que foi configurado na página do console do Google. Mas para encapsular a classe do `GoogleCloudMessaging`, a fim de tornar o código mais simples, criei a classe `GCM`, a qual você pode reutilizar em todos os seus projetos.



GCM.java

```
public class GCM {
    private static final String TAG = "gcm";
    public static final String PROPERTY_REG_ID = "registration_id";
    // Preferências para salvar o registration id
    private static SharedPreferences getGCMPreferences(Context context) {
        SharedPreferences sharedPreferences = context.getSharedPreferences("livroandroid_gcm",
            Context.MODE_PRIVATE);
        return sharedPreferences;
    }
    // Retorna o registration id salvo nas prefs
    public static String getRegistrationId(Context context) {
        final SharedPreferences prefs = getGCMPreferences(context);
        String registrationId = prefs.getString(PROPERTY_REG_ID, "");
        if (registrationId == null || registrationId.trim().length() == 0) {
            return null;
        }
        return registrationId;
    }
    // Salva o registration id nas prefs
    private static void saveRegistrationId(Context context, String registrationId) {
        final SharedPreferences prefs = getGCMPreferences(context);
        SharedPreferences.Editor editor = prefs.edit();
        editor.putString(PROPERTY_REG_ID, registrationId);
        editor.commit();
    }
    // Faz o registro no GCM
    public static String register(Context context, String projectNumber) {
        GoogleCloudMessaging gcm = GoogleCloudMessaging.getInstance(context);
```

```

    try {
        Log.d(TAG, ">> GCM.registrar(): " + projectNumber);
        String registrationId = gcm.register(projectNumber);
        if(registrationId != null) {
            // Salva nas prefs
            saveRegistrationId(context, registrationId);
        }
        Log.d(TAG, "<< GCM.registrar() OK, registration id: " + registrationId);
        return registrationId;
    } catch (IOException e) {
        Log.e(TAG, "<< GCM.registrar() ERRO: " + e.getMessage(), e);
    }
    return null;
}

// Cancelar o registro no GCM
public static void unregister(Context context) {
    GoogleCloudMessaging gcm = GoogleCloudMessaging.getInstance(context);
    try {
        gcm.unregister();
        saveRegistrationId(context, null);
        Log.d(TAG, "GCM cancelado com sucesso.");
    } catch (IOException e) {
        Log.e(TAG, "GCM erro ao desregistrar: " + e.getMessage(), e);
    }
}
}
}

```

A seguir, temos a explicação dos métodos da classe GCM.

```
public static String register(Context context, String projectNumber)
```

Faz o registro no GCM e retorna o registration id. Este método precisa ser chamado dentro de uma thread. Depois de obter o registration id, ele é salvo automaticamente nas preferências com a classe SharedPreferences.

```
public static void unregister(Context context)
```

Cancela o registro no GCM e limpa as preferências. Este método precisa ser chamado dentro de uma thread.

```
public static String getRegistrationId(Context context)
```

Retorna o registration id salvo nas preferências. Se o registration id estiver salvo, significa que o aplicativo está registrado no GCM. Se o retorno for nulo, significa que o dispositivo ainda não está registrado.

A seguir, vamos ver alguns exemplos de código de como utilizar esta classe. Para o registro no GCM, fariamos assim:

```
// Faz o registro no GCM
String registrationId = GCM.register(getContext(), "Project Number AQUI");
```

Para verificar se o dispositivo já está registrado, basta chamar o método `getRegistrationId(context)`. Se o retorno for nulo, significa que o dispositivo ainda não está registrado.

```
// Recupera o registration id
final String registrationId = GCM.getRegistrationId(this);
```

E, por último, para cancelar o registro, basta fazer:

```
// Faz o registro no GCM
GCM.unregister(getContext());
```

Utilizar a classe GCM é simples. Note que é necessário informar o **Project Number** para fazer o registro. Portanto, crie no projeto uma interface ou classe abstrata chamada `Constants` com a constante `PROJECT_NUMBER`.

 **Constants.java**

```
package br.livro.android.cap28.push;
public interface Constants {
    // Project Number criado na página do Google Developers Console
    String PROJECT_NUMBER = "999559052058";
}
```

É costume deixar o **Project Number** em uma constante, pois vamos utilizá-lo em outros lugares. Sendo assim, futuramente no código poderemos utilizar essa constante para fazer o registro.

```
// Faz o registro no GCM
String registrationId = GCM.register(getContext(), Constants.PROJECT_NUMBER);
```

28.9 Configurando o projeto Android

Agora vamos para a pior parte, que é configurar o arquivo de manifesto. Como a configuração é meio trabalhosa, recomendo que você copie o arquivo *AndroidManifest.xml* dos exemplos do livro para não correr o risco de digitar algo errado.

Caso tenha utilizado um nome de pacote diferente de `br.com.livroandroid.hellopush`, substitua o pacote nos lugares indicados. O código-fonte a seguir está comentado, portanto leia atentamente o significado de cada configuração:



AndroidManifest.xml

```

<?xml version="1.0" encoding="utf-8"?>
<manifest package="br.com.livroandroid.hellopush" . . .>
    <!-- Para receber a mensagem precisa de internet -->
    <uses-permission android:name="android.permission.INTERNET" />
    <!-- O GCM precisa se conectar a uma conta do Google. -->
    <uses-permission android:name="android.permission.GET_ACCOUNTS" />
    <!-- Permissão utilizada para travar a tela, e evitar o modo de espera. -->
    <uses-permission android:name="android.permission.WAKE_LOCK" />
    <!-- Permissão customizada necessária para receber as mensagens.
        Ela precisa ser chamada PACOTE.permission.C2D_MESSAGE -->
    <permission android:name="br.com.livroandroid.hellopush.permission.C2D_MESSAGE"
        android:protectionLevel="signature" />
    <uses-permission android:name="br.com.livroandroid.hellopush.permission.C2D_MESSAGE" />
    <!-- Declara a permissão para se registrar no GCM e receber mensagens -->
    <uses-permission android:name="com.google.android.c2dm.permission.RECEIVE" />
    <application . . . >
        <!-- Versão do Google Play Services. -->
        <meta-data
            android:name="com.google.android.gms.version"
            android:value="@integer/google_play_services_version" />
        <activity android:name=".MainActivity" . . .>
            <!-- Tudo igual como sempre aqui -->
        </activity>
        <!-- BroadcastReceiver para receber as mensagens do GCM, por meio de intents.
            Este receiver precisa estar declarado no manifesto, para que as mensagens sejam
            interceptadas mesmo com o aplicativo fechado. Este receiver vai iniciar o service
            GcmIntentService, que está declarado mais abaixo. -->
        <receiver android:name=".GcmBroadcastReceiver"
            android:permission="com.google.android.c2dm.permission.SEND" >
            <intent-filter>
                <!-- Filtrar as ações para receber mensagens. -->
                <action android:name="com.google.android.c2dm.intent.RECEIVE" />
                <category android:name="br.com.livroandroid.hellopush" />
            </intent-filter>
        </receiver>
        <!-- Service chamado pelo receiver acima. Deve conter o código para ler as mensagens. -->
        <service android:name=".GcmIntentService" />
    </application>
</manifest>

```

No manifesto, declaramos a classe `MainActivity` como de costume, mais um broadcast receiver chamado `GcmBroadcastReceiver` e um service chamado `GcmIntentService`.

A seguir, podemos ver o código do broadcast receiver que tem como objetivo interceptar as mensagens de push. Esse receiver está interceptando a ação `com.google.android.c2dm.intent.RECEIVE`, a qual é enviada por broadcast pelo sistema quando chega uma nova mensagem de push.

GcmBroadcastReceiver.java

```
public class GcmBroadcastReceiver extends WakefulBroadcastReceiver {
    private static final String TAG = "gcm";
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.i(TAG, "GcmBroadcastReceiver.onReceive: " + intent.getExtras());
        // Inicia o service com WAKE LOCK.
        ComponentName comp = new ComponentName(context.getPackageName(),
            GcmIntentService.class.getName());
        startWakefulService(context, (intent.setComponent(comp)));
        setResultCode(Activity.RESULT_OK);
    }
}
```

Repare que esse receiver herda de `WakefulBroadcastReceiver`, um tipo especial de receiver que permite bloquear o processador (wake lock) para ter certeza de que a aplicação vai tratar a mensagem de push. Justamente por isso, colocamos a permissão `WAKE_LOCK` no arquivo de manifesto. Depois de interceptar a mensagem, o receiver inicia um serviço em segundo plano para tratá-la. Isso é feito com o método `startWakefulService(context, intent)`, no qual a intent refere-se à classe `GcmIntentService`, que é o serviço declarado no *AndroidManifest.xml*.

```
<service android:name=".GcmIntentService" />
```

A classe `GcmIntentService` é quem vai definitivamente tratar a mensagem, e o motivo pelo qual a utilizamos é porque serviços podem executar por um longo período de tempo, enquanto receivers precisam executar rapidamente. Apenas para lembrar, esta é uma solução comum em aplicações Android: utilizar o receiver para interceptar as intents e delegar o trabalho para um serviço.

A seguir, podemos visualizar o código-fonte da classe `GcmIntentService`. Ela é filha da classe `IntentService` que estudamos no capítulo 27, sobre serviços.

GcmIntentService.java

```
public class GcmIntentService extends IntentService {
    private static final String TAG = "gcm";
    public GcmIntentService() {
```

```

        super(Constants.PROJECT_NUMBER);
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        Bundle extras = intent.getExtras();
        Log.i(TAG, "GcmIntentService.onHandleIntent: " + extras);
        GoogleCloudMessaging gcm = GoogleCloudMessaging.getInstance(this);
        // Verifica o tipo da mensagem
        String messageType = gcm.getMessageType(intent);
        if (!extras.isEmpty()) {
            // O extras.isEmpty() precisa ser chamado para ler o Bundle
            // Verifica o tipo da mensagem, no futuro podemos ter mais tipos
            if (GoogleCloudMessaging.MESSAGE_TYPE_SEND_ERROR.equals(messageType)) {
                // Erro
                onError(extras);
            } else if (GoogleCloudMessaging.MESSAGE_TYPE_MESSAGE.equals(messageType)) {
                // Mensagem do tipo normal. Faz a leitura do parâmetro "msg"
                // enviado pelo servidor
                onMessage(extras);
            }
        }
        // Libera o wake lock, que foi bloqueado pela classe
        // "GcmBroadcastReceiver".
        GcmBroadcastReceiver.completeWakefulIntent(intent);
    }

    private void onError(Bundle extras) {
        Log.d(TAG, "Erro: " + extras.toString());
    }

    private void onMessage(Bundle extras) {
        // Lê a mensagem e mostra uma notificação
        String msg = extras.getString("msg");
        Log.d(TAG, msg);
        Intent intent = new Intent(this, MainActivity.class);
        intent.putExtra("msg", msg);
        NotificationUtil.create(this, 1, intent, R.drawable.ic_notification_icon, "Nova
mensagem", msg);
    }
}

```

Importante: o construtor da classe `GcmIntentService` recebe como parâmetro a constante `Constants.PROJECT_NUMBER`, que é o Project Number que copiamos na página de console do Google.

No método `onHandleIntent(intent)` da classe `GcmIntentService` estamos lendo a mensagem de push que chega via a intent. O código está verificando o tipo da mensagem e chamando os métodos `onError(bundle)`, no caso de erro, ou `onMessage(bundle)`, no caso de sucesso. Atente-se para os tipos das mensagens, pois no futuro o GCM pode ser expandido e novos tipos poderão surgir.

Um fator importante sobre a lógica do código-fonte é que no final do método `onHandleIntent(intent)` é chamado o método `GcmBroadcastReceiver.completeWakefulIntent(intent)`. Isso é feito para liberar o bloqueio do processador (wake lock), o qual foi travado no passo anterior pelo receiver `GcmBroadcastReceiver`.

No método `onMessage(bundle)` recebe-se a mensagem. Neste caso estamos lendo um parâmetro "msg" do tipo `String`, pois o servidor que vamos criar vai enviar uma mensagem exatamente com esse parâmetro. Logo depois de ler a mensagem, o aplicativo vai mostrar uma notificação para o usuário. Ao clicar na notificação, a `MainActivity` será aberta para ler a mensagem.

```
private void onMessage(Bundle extras) {
    // Lê a mensagem e mostra uma notificação
    String msg = extras.getString("msg");
    Log.d(TAG, msg);
    Intent intent = new Intent(this, MainActivity.class);
    intent.putExtra("msg", msg);
    NotificationUtil.create(this, 1, intent, R.drawable.ic_notification_icon,
        "Nova mensagem", msg);
}
```

28.10 Criando a activity para fazer o registro no GCM

Estamos quase terminando o código do projeto Android, falta apenas a activity.

No arquivo de layout, vamos inserir dois botões na tela, **Registrar** e **Cancelar**. O botão **Registrar** vai fazer o registro no GCM e obter o **registration id** do dispositivo. O botão **Cancelar** obviamente vai cancelar o registro. Na tela, também vamos inserir um `TextView` para mostrar as mensagens que vamos receber por push.



/res/layout/activity_main.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="16dp" android:orientation="vertical">
    <LinearLayout android:orientation="vertical"
        android:layout_width="match_parent" android:layout_height="wrap_content">
```

```

<Button
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:text="Registrar" android:onClick="onClickRegistrar" />
<Button
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:text="Cancelar" android:onClick="onClickCancelar" />
</LinearLayout>
<TextView android:id="@+id/text"
    android:layout_marginTop="28dp" android:text="Clique em registrar."
    android:layout_width="wrap_content" android:layout_height="wrap_content" />
</LinearLayout>

```

Com o layout pronto, estamos a um passo de terminar o exemplo. Falta apenas o código da activity, que podemos visualizar a seguir.

MainActivity.java

```

public class MainActivity extends AppCompatActivity {
    private static final String TAG = "livroandroid";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        // Verifica se o Google Play Services está instalado
        boolean ok = checkPlayServices();
        if (ok) {
            // Já está registrado
            String regId = GCM.getRegistrationId(this);
            setText(regId);
            // Quando iniciar, lê a msg da notificação
            String msg = getIntent().getStringExtra("msg");
            setText(msg);
        }
    }
    @Override
    protected void onNewIntent(Intent intent) {
        super.onNewIntent(intent);
        // Chamado ao clicar na notificação se a activity já estava aberta
        // Devido à configuração android:launchMode="singleTop"
        // Lê a msg da notificação
        String msg = intent.getStringExtra("msg");
        setText(msg);
    }
    private Context getContext() {

```

```

        return this;
    }

    // Mostra msg de debug na tela
    private void setText(Final String s) {
        if(s != null) {
            runOnUiThread(new Runnable() {
                @Override
                public void run() {
                    TextView text = (TextView) findViewById(R.id.text);
                    text.setText(s);
                    Log.d(TAG, s);
                }
            });
        }
    }

    // Faz o registro no GCM
    public void onClickRegistrar(View view) {
        new Thread() {
            @Override
            public void run() {
                super.run();
                String regId = GCM.getRegistrationId(getApplicationContext());
                if (regId == null) {
                    // Faz o registro e pega o registration id
                    regId = GCM.register(getApplicationContext(), Constants.PROJECT_NUMBER);
                    setText("Registrado com sucesso.\n" + regId);
                } else {
                    toast("Você já está registrado.");
                }
            }
        }.start();
    }

    // Cancela o registro no GCM
    public void onClickCancelar(View view) {
        new Thread() {
            @Override
            public void run() {
                super.run();
                GCM.unregister(getApplicationContext());
                setText("Cancelado com sucesso!");
            }
        }.start();
    }
}

```

```
// Verifica se o Google Play Services está instalado
private boolean checkPlayServices() {
    int resultCode = GooglePlayServicesUtil.isGooglePlayServicesAvailable(this);
    if (resultCode != ConnectionResult.SUCCESS) {
        if (GooglePlayServicesUtil.isUserRecoverableError(resultCode)) {
            int PLAY_SERVICES_RESOLUTION_REQUEST = 9000;
            GooglePlayServicesUtil.getErrorDialog(resultCode, this,
                PLAY_SERVICES_RESOLUTION_REQUEST).show();
        } else {
            toast("Este dispositivo não suporta o Google Play Services.");
            finish();
        }
        return false;
    }
    return true;
}

private void toast(final String s) {
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            Toast.makeText(getContext(), "Toast: " + s, Toast.LENGTH_SHORT).show();
        }
    });
}
}
```

Como a MainActivity pode ser aberta por uma notificação, adicione o atributo `android:launchMode="singleTop"` à sua configuração.

```
<activity android:name=".MainActivity" android:launchMode="singleTop" . . . />
```

Apenas lembrando, essa configuração faz com que o Android crie apenas uma instância da activity, sendo que, se o usuário clicar na notificação com a activity aberta, ela será reutilizada e o método `onNewIntent(intent)` será chamado. Sem esse atributo, ao clicar na notificação, sempre uma nova activity seria criada e colocada na pilha.

E isso é tudo! Agora você já sabe como deve ser feito o código-fonte do aplicativo Android para receber as mensagens de push. Se você fez tudo passo a passo, execute o aplicativo para se registrar no GCM. Depois, com o **registration id** do dispositivo em mãos, podemos enviar uma mensagem por push utilizando o código Java que já mostramos.

Dica: no código da `activity`, estamos verificando se o Google Play Services está instalado no dispositivo. Caso não esteja, não é possível fazer o registro no GCM. Nesse caso, para simplificar o exemplo, estamos chamando o método `finish()` a fim de fechar o aplicativo. Outro detalhe importante no código é que estamos utilizando `threads` (ou `async task`) para se comunicar com o servidor do GCM. Isso é necessário. Eu utilizo `threads` simples nos exemplos para facilitar a leitura. Mas na prática costumo utilizar o método `startTask(task)`, que encapsula a `AsyncTask`, conforme fizemos no projeto dos carros.

28.11 Links úteis

Neste capítulo, estudamos o GCM e aprendemos a enviar mensagens de push para o aplicativo. Existem muitas particularidades deste serviço que você precisa conhecer, por isso, nada melhor do que finalizar seus estudos lendo a documentação oficial.

- **Google Services – Google Cloud Messaging**

<https://developer.android.com/google/gcm/index.html>



CAPÍTULO 29

Salvando o estado da aplicação

Entender o ciclo de vida de uma activity ou fragment é um fator decisivo para construir um aplicativo de sucesso.

Neste capítulo, vamos entender quando uma activity pode ser destruída e re-criada, e como o aplicativo deve salvar o estado das informações. Provavelmente você já conhece o conceito, pois já mostramos casos simples sobre como salvar informações durante a troca de configurações de sistema, mas este capítulo visa complementar os seus estudos.

Se você quiser, pule este capítulo. Eu só preciso garantir que você domine o assunto, pois missão dada é missão cumprida.

29.1 Troca de configurações – configuration changes

Quando uma activity é criada, o método `onCreate(bundle)` é chamado, e na sequência todos os métodos do ciclo de vida são chamados, até chegar a vez do `onDestroy()`, quando a activity é destruída. Sabemos que uma activity será destruída quando o usuário pressionar o botão voltar ou o aplicativo chamar o método `finish()`.

Porém, existem outros fatores que podem fazer com que uma activity seja destruída, que é quando ocorre uma troca de configuração do sistema, chamada de *configuration changes*. A lista a seguir mostra as trocas de configurações do sistema mais comuns:

1. Troca de orientação (vertical e horizontal) – acontece ao girar o dispositivo.
2. Abrir o teclado físico em dispositivos com teclado.
3. Troca de idioma nas configurações do Android.

Esses três casos são exemplos de troca de configuração, e, se alguma dessas condições ocorrer, o Android vai encerrar a activity atual e vai recriá-la logo em seguida para que a nova configuração faça efeito.

Veja que até que faz sentido. Vamos dizer que você tenha um layout para vertical e horizontal, conforme demonstrado a seguir:

- `/res/layout/activity_main.xml`
- `/res/layout-land/activity_main.xml`

Neste caso, o identificador `land` é de `landscape` e representa o layout horizontal. O sistema do Android vai utilizar o layout correto conforme a orientação do dispositivo. Quando a activity é criada pela primeira vez na vertical (`portrait`), o layout `/res/layout/activity_main.xml` é utilizado. Então, ao girar a tela para horizontal (`landscape`), o Android vai destruir essa activity e recriá-la novamente, mas desta vez quando o método `onCreate(bundle)` for chamado o layout utilizado será o `/res/layout-land/activity_main.xml`. O Android apresenta esse comportamento porque muitas vezes é preciso atualizar o layout em casos de alterações nas configurações do sistema.

Com os fragments, o conceito é o mesmo, e o método `onCreateView()` pode inflar uma view diferente dependendo da orientação. É justamente por isso que a activity e seus fragments são destruídos e recriados, assim você tem uma chance de recriar a tela com as configurações adequadas.

Como desenvolvedor, você deve estar ciente desse comportamento e conhecer como salvar o estado da aplicação nesses casos.

29.2 Salvando o estado com o método `onSaveInstanceState(bundle)`

Para testar o próximo exemplo, abra o projeto deste capítulo e execute-o no emulador. O código é parecido com um que já fizemos ao estudar fragments, mas agora vamos relembrar somente a questão do ciclo de vida e o método `onSaveInstanceState(bundle)`.



Fragment1.java

```
public class Fragment1 extends DebugFragment {
    private int count;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_1, container, false);
        final TextView t = (TextView) view.findViewById(R.id.text);
```

```
// Recupera o estado
if(savedInstanceState != null) {
    count = savedInstanceState.getInt("count");
    // Atualiza o TextView com o valor salvo
    t.setText("Count: " + count);
}

view.findViewById(R.id.btOk).setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        count++;
        t.setText("Count: " + count);
    }
});

return view;
}

@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    // Salva o estado
    outState.putInt("count", count);
}
}
```

Nota: a classe `Bundle` é como uma `HashMap` e salva os objetos por chave e valor. Ela pode salvar vários tipos, como `Integer`, `Boolean`, `String`, `Serializable`, `Parcelable` etc.

Esse fragment tem um simples layout com um botão que fica incrementando o contador. Note que este ele é filho de `DebugFragment`. A activity desse fragment apenas insere o fragment no layout, sendo que ela é filha de `DebugActivity`.

Estou fazendo essas heranças porque quero reforçar a ordem da chamada dos métodos do ciclo de vida.

Portanto, faça o seguinte teste:

1. Execute o projeto, e clique no botão **Incrementar** cinco vezes.
2. O valor do contador será atualizado para 5.
3. Gire a tela do dispositivo (no emulador é **Ctrl+F11**).
4. Depois de girar a tela, o valor do contador deve continuar como 5.

A figura 291 mostra o resultado. O método `onSaveInstanceState(bundle)` salvou o estado do aplicativo, e o valor do contador é lido na inicialização do fragment.

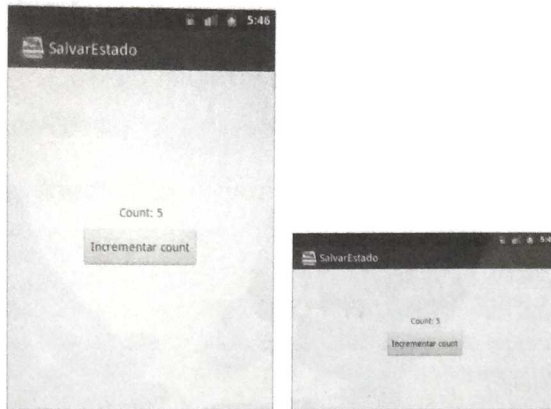


Figura 29.1 – Salvando o estado do fragment.

O mais interessante desse exemplo são os logs do LogCat, que imprimem a ordem correta de execução dos métodos da activity e do fragment. Veja que o método `onSaveInstanceState(bundle)` é chamado sempre antes do `onPause()`. Os logs a seguir são impressos ao girar o dispositivo:

```
// Salvando estado
Fragment1.onSaveInstanceState()
DemoSalvarEstadoActivity.onSaveInstanceState()
Fragment1.onPause()
DemoSalvarEstadoActivity.onPause()
Fragment1.onStop()
DemoSalvarEstadoActivity.onStop()
Fragment1.onDestroyView()
Fragment1.onDestroy()
Fragment1.onDetach()
DemoSalvarEstadoActivity.onDestroy()
// Recriando a activity e seus fragments
Fragment1.onCreate()
DemoSalvarEstadoActivity.onCreate()
Fragment1.onActivityCreated()
Fragment1.onStart()
DemoSalvarEstadoActivity.onStart()
DemoSalvarEstadoActivity.onResume()
Fragment1.onResume()
```

Ao fazer esse teste de girar a tela, você pode ver claramente como os métodos do ciclo de vida são chamados e como o Android faz para destruir a activity e

recriá-la. Saiba que o método `onSaveInstanceState()` também existe na `activity`, mas atualmente é recomendado deixar a lógica dentro dos `fragments` devido à possibilidade de reutilizar o código, por isso estou mostrando como salvar o estado somente com o `fragment`.

Nota: caso o método `finish()` seja chamado explicitamente no código da `activity`, estamos conscientemente fechando a tela, e dessa forma o Android não chamará o método `onSaveInstanceState(bundle)`. O método `onSaveInstanceState(bundle)` será chamado apenas quando ocorrer uma troca de configuração de sistema.

29.3 Salvando o estado com o método `setRetainInstance(boolean)`

Salvar os dados do `fragment` com o método `onSaveInstanceState(bundle)` é simples, mas você precisa preencher o `Bundle` manualmente. Caso você queira fazer algo ainda mais simples, é possível manter a instância do `fragment` viva durante a troca de configuração, assim os dados serão preservados automaticamente.

Se o método `setRetainInstance(true)` for chamado no `fragment`, algo que geralmente é feito no método `onCreate(bundle)`, a instância do objeto do `fragment` será preservada em memória, e apenas a `view` será criada novamente.

O código a seguir está atualizado para salvar o estado com o método `setRetainInstance(true)`. Veja que é muito simples. Por sorte, como neste caso a variável é inteira, ela é inicializada com 0 (zero), sendo assim nem precisamos validar se ela está inicializada. Ao girar a tela, o estado do contador será mantido normalmente.



Fragment1.java

```
public class Fragment1 extends CicloVidaFragment {
    private int count;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Mantém o fragment vivo
        setRetainInstance(true);
    }
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_1, container, false);
```



```

final TextView t = (TextView) view.findViewById(R.id.text);
// O atributo count vai permanecer em memória
t.setText("Count: " + count);
view.findViewById(R.id.btOk).setOnClickListener(new OnClickListener() {
    @Override
    public void onClick(View v) {
        count++;
        t.setText("Count: " + count);
    }
});
return view;
}
)

```

29.4 A importância de reter a instância do fragment

Até aqui foi uma revisão, pois já estudamos esse assunto no capítulo 8, sobre fragments, então agora vamos complicar um pouco.

Estas duas abordagens fazem a mesma coisa, mas o método `setRetainInstance(true)` é mais simples, pois com ele não é preciso salvar o estado manualmente preenchendo o Bundle, pois a instância do fragment é preservada. Mas existe outra grande vantagem do método `setRetainInstance(true)`, que é quando temos alguma thread executando para buscar informações.

Vamos imaginar que temos um aplicativo que busca notícias de um web service. Se o usuário girar o dispositivo, o que vai acontecer? Sabemos que por padrão o Android vai destruir e recriar a activity, junto com seus fragments. Sendo assim, o que acontece com a thread que estava executando?

Se o desenvolvedor não controlar isso, a thread vai continuar executando, mas a activity e fragment serão destruídos durante a troca de orientação. Uma nova activity e fragment serão criados, e você precisará buscar os dados no web service novamente. Isso significa que, se você não projetar seu código de maneira adequada, sempre que você girar a tela, a thread/tarefa será executada novamente, pois o Android vai recriar a activity.

Dentro desse contexto, reter a instância de um fragment com o método `setRetainInstance(true)` pode ajudar a resolver o problema, pois ao manter o objeto vivo as threads que executam dentro do fragment também serão preservadas. Então neste caso você pode controlar se a thread já foi iniciada ou até se o resultado já foi obtido, e no máximo será necessário recriar a view do fragment no método `onCreateView()`.

Lembre-se de que, durante a troca de orientação, mesmo se você reter a instância do fragment, o método `onCreateView()` continua sendo chamado, pois o fragment precisa recriar a sua view. Nesse momento você pode validar o estado em que se encontram os dados, e decidir se é necessário iniciar a busca de web services ou não, caso o aplicativo já tenha as informações.

29.5 Manter uma thread executando durante a troca de orientação

No capítulo 7, sobre views, estudamos a classe `ProgressBar` que pode ser utilizada para mostrar uma animação que fica girando, ou uma barra que pode incrementar o progresso de uma tarefa. Isso depende do estilo que for utilizado.

O código a seguir mostra como criar um `ProgressBar` com uma barra horizontal:

```
<ProgressBarandroid:id="@+id/barraProgresso" style="?android:attr/progressBarStyleHorizontal"
    android:layout_width="match_parent" android:layout_height="wrap_content"
    android:max="100" android:progress="0" />
```

Nota: o próximo exemplo é avançado, leia quando achar necessário.

A figura 29.2 mostra o exemplo que fizemos no capítulo 7, que apenas inicia uma thread que fica incrementando um contador de 0 a 100 e atualizando o valor do progresso.

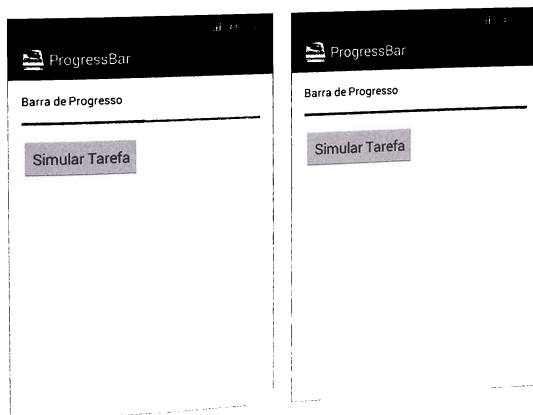


Figura 29.2 – Thread executando.

Mas o que acontece se você girar a tela durante o andamento dessa tarefa? Sabemos que a *activity* vai ser destruída e recriada, então o que acontece com a *thread* que estava executando?

Não existe nada que vai parar a *thread* durante a troca de orientação; assim, caso você queira parar a *thread*, você deve interrompê-la no método `onPause()` ou `onStop()` do ciclo de vida da *activity*. Mas neste caso não queremos interromper a *thread*, e sim deixá-la executando, pois a tarefa precisa continuar independentemente da troca de orientação.

Portanto, no exemplo do capítulo 7, ao girar a tela, a *activity* é destruída, porém a *thread* continua viva e executando. Mas como essa *thread* está associada com uma *activity* que será destruída, se ela atualizar a *view* nada vai acontecer. Quando a nova *activity* for criada, o *ProgressBar* apenas vai mostrar o último valor que recebeu a atualização, pois as *views* do Android internamente também salvam seu próprio estado. Porém, a *thread* se perde no meio do caminho, e o problema é que ela está associada com uma *activity* que já não está mais sendo mostrada ao usuário. Inclusive isso é um problema, pois devido a essa referência a *activity* antiga não pode ser eliminada de memória pelo *garbage collector* e tivemos um *memory leak*. Mas esse é um assunto mais avançado, e vamos deixar pra lá.

O que nós queremos é deixar a *thread* viva, mas, assim que a nova *activity* e *view* forem criadas, queremos atualizar o progresso da tarefa, como se nada tivesse acontecido. Tudo deve funcionar perfeitamente, mesmo se o usuário ficar brincando de girar o celular.

O modo indicado de solucionar esse problema é utilizar um *fragment* e reter a instância com o método `setRetainInstance(true)`. Feito isso, a *thread* dentro do *fragment* pode continuar executando sem problemas, e assim que a *view* do *fragment* for criada depois da troca de orientação podemos atualizar o status do progresso. A *thread* permanece viva, e apenas o *fragment* vai mostrar uma *view* diferente.

O código a seguir mostra a solução para esse problema, e com ele vamos conseguir girar a tela sem parar a tarefa e a atualização do progresso.

 **ProgressBarDemoActivity.java**

```
public class ProgressBarDemoActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_progress_bar_demo);
        if(savedInstanceState == null) {
            getSupportFragmentManager().beginTransaction().add(R.id.layoutFrag,
```

```

        new ProgressBarDemoFragment().commit();
    }
}
}

```

 /res/layout/activity_progress_bar_demo.xml

```

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="@dimen/activity_horizontal_margin"
    android:id="@+id/layoutFrag">

</FrameLayout>

```

Como você pode ver, a activity não faz nada, e mais uma vez vamos delegar a lógica para o fragment.

 ProgressBarDemoFragment.java

```

public class ProgressBarDemoFragment extends Fragment {
    private int progress;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true); // Deixa o fragment em memória
    }
    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
        @Nullable Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_progress_bar_demo, container, false);
        Button b = (Button) view.findViewById(R.id.btOK);
        b.setOnClickListener(new Button.OnClickListener() {
            @Override
            public void onClick(View view) {
                executarTarefa();
            }
        });
        return view;
    }
    private void executarTarefa() {
        new Thread(new Runnable() {
            public void run() {
                // A variável progress fica em memória
                // O for começa em 0 na primeira vez ou continua de onde parou
                for (int i = progress; i <= 100; i++) {

```



```

        // Salva o estado do int progress
        progress = i;
        updateProgressBar();
        try {
            Thread.sleep(50);
        } catch (InterruptedException e) {
        }
    }
    // Ao terminar a tarefa zera a variável do progresso
    progress = 0;
    Log.d("livroandroid", "Fim");
}
}).start();
}
// Atualiza a barra de progresso
private void updateProgressBar() {
    // Somente atualiza a view se o fragment está vivo e vinculado a uma activity
    if(getView() != null && !isDetached()) {
        getView().post(new Runnable() {
            public void run() {
                // Barra de Progresso
                final ProgressBar progressBar = (ProgressBar)
                    getView().findViewById(R.id.barraProgresso);
                Log.d("livroandroid", ">>> Progress: " + progress);
                progressBar.setProgress(progress);
            }
        });
    }
}
}
}
}

```

O arquivo de layout do fragment tem o mesmo layout do exemplo do `ProgressBar` que fizemos no capítulo 7, portanto verifique nos exemplos do livro. O layout apenas contém o `ProgressBar` e o botão para iniciar a tarefa.

Se você executar esse exemplo e girar a tela, verá que a tarefa continuará executando, pois o fragment foi retido em memória, de forma que a thread permaneceu viva e associada com o fragment. A variável inteira `progress` ficou em memória, para continuar a simulação da tarefa de onde parou.

O primeiro segredo do exemplo é o loop do `for` não começar em `i=0`, e sim em `i=progress`, pois queremos continuar a atualização do `ProgressBar` de onde parou antes da rotação da tela, e a variável `progress` vai manter o seu estado.

O segundo segredo do exemplo é tomar cuidado para não atualizar a view do fragment, caso ele esteja com um estado inválido. Lembre-se de que, como a activity e o fragment serão destruídos durante a troca de orientação, vai chegar um momento em que a view do fragment ficará nula, pois o fragment será desassociado da activity. Isso acontece depois que os métodos `onDestroyView()` e `onDetach()` são chamados. Mas como a thread continua executando, temos de fazer um teste de condição e validar se a view do fragment existe e se o fragment está associado com a activity. Portanto, adicionei esta condição no código:

```
if(getView() != null && !isDetached()) {  
    getView().post(new Runnable() {  
        // O fragment está ok, podemos atualizar a view  
    });  
}
```

Se você entendeu o que aconteceu, ótimo, pois esse é um dos grandes benefícios de se reter a instância de um fragment. Se você não entendeu neste momento, sem problemas, mas quando achar necessário revise este exercício, pois com certeza isso vai lhe ser muito útil algum dia.

Nota: o método `getView().post(runnable)` utiliza o handler que está dentro da view para executar este código na UI Thread. Outro atalho bem conhecido é o método `runOnUiThread(runnable)` da classe `Activity`. Ambas as classes `Activity` e `View` estão associadas com um handler, que por sua vez está associado com a UI Thread e gerencia uma fila de mensagens. Lembre-se de que você não pode atualizar a view dentro uma thread diferente da UI Thread, por isso esses métodos são necessários.

29.6 Como bloquear a troca de orientação

Muitos aplicativos para smartphone optam por travar a orientação na vertical, o que era uma prática comum antigamente, mas atualmente isso é desencorajado pelas boas práticas do Google.

Mas em todo o caso, para travar a orientação da activity na vertical, utilize o atributo `android:screenOrientation` no manifesto.

```
<activity android:name=".ClasseActivityAqui" android:screenOrientation="portrait" />
```

Por padrão, se você não fizer isso, o Android vai permitir as orientações vertical e horizontal. Para forçar que uma activity fique apenas na horizontal, basta utilizar a constante `landscape`.

```
<activity android:name=".ClasseActivityAqui" android:screenOrientation="landscape" />
```

Nota: atualmente, segundo as boas práticas do Google, é recomendável deixar o aplicativo girar a vontade, pois o usuário deve poder escolher qual a orientação que ele prefere ao utilizar o aparelho. Lembre-se de que é o usuário quem manda no celular dele, e não você.

Outra maneira de forçar a orientação de uma *activity* é utilizar o método `setRequestedOrientation(orientation)` da classe *Activity* dinamicamente no código; porém, é recomendado que essas configurações estejam declaradas no arquivo de manifesto.

Na maioria das vezes, o fato de o usuário girar a tela não deve ser um problema para você; basta salvar o estado das informações e pronto. Às vezes será preciso customizar telas na horizontal, como por exemplo: uma galeria de fotos ou um gráfico. Nesse caso, vale a pena criar dois layouts para melhorar a experiência do usuário na horizontal. Lembre-se de que para isso basta inserir o layout na pasta `/res/layout-land`, e o Android se encarrega de tudo.

29.7 Dispositivos com teclado

Smartphones com teclado físico como o Motorola Milestone também sofrem do problema da troca de configuração, que neste caso pode ser a abertura ou fechamento do teclado físico.

Sempre que o usuário abrir ou fechar o teclado, a *activity* será destruída e recriada da mesma forma que ao girar a tela. Então tudo que você aprendeu sobre salvar o estado da aplicação também se aplica neste caso.

Lembro-me até hoje de quando fui fazer uma demonstração na Motorola e meu aplicativo travou quando o gerente abriu o teclado do seu Milestone. Na época eu não conhecia esse comportamento e também não tinha me preocupado em salvar corretamente o estado da tela.

29.8 Configuração android:configChanges e o método onConfigurationChanged

Até o momento, vimos como salvar e recuperar o estado da tela, para solucionar o problema de o Android destruir a *activity* em certos casos.

Mas e se fosse possível não destruir a *activity*? E se fosse possível que os objetos permanecessem em memória?

Sim, isso é possível. Existe a possibilidade de controlar essas alterações de configurações de sistema manualmente, e sem a necessidade de destruir a *activity*.

Mas neste caso seremos os responsáveis por alterar o layout e atualizá-lo, tudo manualmente. A vantagem de ter o controle total é que a *activity* não é destruída e consequentemente os objetos permanecem em memória. Portanto, não é necessário salvar os dados ou buscá-los novamente. Parece uma boa ideia, não é? Se esse for o seu objetivo, você teria de ter uma conversa mais ou menos assim com o nosso simpático sistema operacional do robzinho verde:

Desenvolvedor: “Fala, Android, como está?”

Android: “Opa, tranquilo, no que posso ajudar?”

Desenvolvedor: “Gostaria de lhe pedir para controlar a tela manualmente. Sou um desenvolvedor avançado e gostaria de ter o controle total, tudo bem?”

Android: “Tudo bem, mas a responsabilidade é sua, ok?”

E depois de alguns segundos o Android ainda tenta aconselhá-lo, sendo sua última chance de voltar atrás.

Android: “Espero que você saiba o que está fazendo. Tem certeza?”

E você responde confiantemente:

Desenvolvedor: “Sim, deixa comigo! Obrigado.”

Pronto, está feito o estrago!

Brincadeiras à parte, controlar a troca de orientação ou outras alterações de sistema manualmente pode ser bom ou ruim, dependendo do caso. Mostraremos aqui como fazer e caberá a você decidir quando utilizar.

Eu particularmente não recomendo fazer isso, a não ser que um dia você encontre um bom motivo para fazê-lo!

Primeiramente, é necessário adicionar o parâmetro `android:configChanges="orientation"` dentro da tag `<activity>` para informar ao Android que vamos controlar tudo manualmente. Dessa forma, todas as vezes que houver uma alteração nas configurações de sistema, o método `onConfigurationChanged(configuration)` será chamado na *activity*, para avisá-lo qual configuração foi alterada.

Existem vários eventos de configuração que podem ser monitorados, mas estamos interessados no caso mais comum, o `orientation`, que ocorre ao trocar a orientação do celular. Neste caso, você pode configurar a *activity* com o parâmetro `android:configChanges="orientation"`.

```
<activity android:name=".ClasseActivityAqui" android:configChanges="orientation" />
```

Caso você queira sobrescrever o comportamento tanto da troca de orientação quanto do estado do teclado, isso poderia ser feito assim:

```

<activity android:name=".ClasseActivityAqui"
    android:configChanges="orientation|keyboardHidden " />

```

Feito isso, você deve sobrescrever o método `onConfigurationChanged(configuration)` na `activity`. Se você der uma rápida analisada na classe `Configuration`, verá que é possível descobrir a configuração que mudou: se foi a orientação, idioma (locale) ou o estado do teclado físico (aberto ou fechado).

```

@Override
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);
    // Mudou alguma configuração de sistema, faça sua lógica aqui
    if (cfg.orientation == Configuration.ORIENTATION_PORTRAIT) {
        Log.i(TAG, "Trocou para Vertical");
    } else if (cfg.orientation == Configuration.ORIENTATION_LANDSCAPE) {
        Log.i(TAG, "Trocou para Horizontal");
    }
}

```

O problema é que agora todo o comportamento de que o sistema do Android cuidava automaticamente agora é sua responsabilidade. Se precisar ter um layout diferenciado entre vertical e horizontal, você mesmo será responsável por fazer alguma mágica aqui, pois o método `onCreate(bundle)` não será chamado, até porque a `activity` permanece viva e nada aconteceu.

Para ter uma ideia do tamanho do problema, se você tiver uma tela com um `EditText` e girar a tela, o Android sempre salva o estado deste `EditText`, assim como ele salva o estado de todas as suas `views`. Mas se você sobrescrever esse comportamento o estado não será mais salvo, pois agora isso é responsabilidade sua. Portanto, ao sobrescrever esse comportamento, o Android passa a responsabilidade de atualizar a tela e demais configurações para o desenvolvedor, e isso não é bom na maioria dos casos.

Não vou falar mais muito disso aqui, pois segundo as boas práticas do Android não é recomendado sobrescrever essas configurações, e este breve tópico foi apenas para alertá-lo sobre isso.

Estou explicando isso porque é comum encontrar sites, blogs etc. explicando esse assunto, portanto fica aqui minha dica para você evitar ao máximo utilizar esse recurso, a não ser que algum dia você tenha um bom motivo, e saiba o que está fazendo.

Recomendo salvar o estado do aplicativo utilizando os métodos `onSaveInstanceState(bundle)` e `setRetainInstance(true)` do `fragment`. Isso já foi explicado e conforme vimos não é tão difícil assim.

29.9 Salvando o estado no projeto dos carros

No projeto dos carros, se você girar o dispositivo com a tela da lista de carros aberta, verá que a lista será recarregada, pois a activity foi recriada, portanto o web service executou novamente.

Enfim, este breve tópico foi apenas para lembrá-lo disso, já que falamos sobre salvar o estado da aplicação. Mas salvar o estado no projeto dos carros vou deixar para você, caso queira fazer algum exercício.

29.10 Links úteis

Neste capítulo, aprendemos detalhes importantes sobre como salvar o estado da aplicação. Siga um link da documentação oficial para você complementar seus estudos.

- **Handling Runtime Changes**

<http://developer.android.com/guide/topics/resources/runtime-changes.html>



CAPÍTULO 30

Suportando diferentes tamanhos de telas

Neste capítulo, vamos aprender os conceitos necessários para criar aplicativos que funcionem em diversas telas e resoluções. Você aprenderá o conceito de densidade e a notação dp (density-independent pixels).

Quando estava escrevendo o livro, fiquei na dúvida se colocava este capítulo no início, logo depois de explicar o que é uma view, ou depois como estou fazendo agora. Ao refletir um pouco, cheguei à conclusão de que existem assuntos que são delicados de explicar, e é melhor você ler quando já tiver certo domínio da plataforma.

Este capítulo vai lhe tirar muitas dúvidas, ou vai deixá-lo com muitas dúvidas. Tudo vai depender se você está pronto para lê-lo, ou não. Em minha opinião, entender o assunto deste capítulo é um grande diferencial para qualquer desenvolvedor Android.

30.1 Unidades de medida

Frequentemente, ao definir um arquivo de layout, são utilizados os valores `match_parent` e `wrap_content` para definir a largura e a altura de um componente. Também é possível informar um valor numérico com uma notação indicando a dimensão, como por exemplo 10dp. Dessa forma, é possível controlar exatamente o tamanho que o componente ocupará.

Para definir dimensões no Android, é possível usar as seguintes opções:

Opção	Descrição
px (pixels)	Número de pixels da tela. Nunca utilize a notação px.
in (polegadas)	Baseado no tamanho físico da tela.
mm (milímetros)	Baseado no tamanho físico da tela.

Opção	Descrição (cont.)
pt (pontos)	1/72 de uma polegada, baseado no tamanho físico da tela.
dp	(density-independent pixels) Essa unidade é relativa à densidade da tela. Recomenda-se seu uso para representar tamanhos de largura e altura das views. O compilador aceita os valores “dp” e “dip”.
sp	(scale-independent pixels) Idem ao dp, mas também considera o tamanho da fonte que o usuário está utilizando. É recomendado utilizar essa unidade para especificar o tamanho de uma fonte, para que ela seja automaticamente ajustada conforme a tela do dispositivo.

Nota: nunca utilize a notação px de pixels, pois, dependendo da resolução da tela, será apresentado um tamanho diferente. Sempre utilize a notação dp ou sp. Para tamanhos de fonte, utilize a notação sp.

30.2 Tamanho de tela (screen size)

A medida **screen size** refere-se ao tamanho físico da tela, medida na diagonal. A plataforma separa os tamanhos físicos de tela em quatro diferentes categorias: pequena (*small*), normal (*normal*), grande (*large*) e extragrande (*xlarge*).

Para deixar mais clara a comparação de cada nomenclatura com aparelhos reais, veremos a seguir uma lista que exhibe cada categoria e alguns celulares que se enquadram nela.

- *small* – Podemos inserir neste grupo o celular Sony Ericsson XPeria Mini, que tem uma pequena tela QVGA de 240x320px.
- *normal* – Celulares como o primeiro HTC G1, o HTC Magic e o Motorola DEXT têm uma tela HVGA média de 320x480px. Neste grupo, também se encontram os celulares um pouco maiores e com melhor resolução, como os aparelhos Nexus One, Sony Ericsson XPeria X10, Motorola Milestone e Samsung Galaxy S, os quais têm uma tela WGA que varia entre 480x800px e 480x854px. Hoje em dia, este é o grupo dos smartphones em geral.
- *large* – Tablets de 7” que têm uma tela com aproximadamente 1024x600px.
- *xlarge* – Tablets de 10” que têm uma tela com aproximadamente 1280x800px.

Alerta: essas medidas de tamanhos de tela são aproximadas e podem ter variações.

Baseado nesses tamanhos de tela, chamados de **screen sizes**, podemos criar arquivos específicos para cada um. Por exemplo, podemos customizar a aplicação para um celular com uma tela pequena, utilizando recursos específicos para telas pequenas, como imagens, arquivos de layouts, textos e tudo o que temos direito. Para isso, basta criarmos pastas com a seguinte nomenclatura:

- */res/drawable-small* – Imagens customizadas para telas pequenas.
- */res/layout-small* – Arquivos de layout customizados para telas pequenas.
- */res/values-small* – Arquivos de textos customizados para telas pequenas.

O Android vai tomar conta do resto, e, se existir um arquivo específico para o tamanho de tela do dispositivo, esse arquivo será utilizado em vez do arquivo padrão. Neste caso do tamanho da tela, é comum ter de customizar o layout para telas pequenas com a pasta */res/layout-small*, pois muitas vezes o conteúdo não cabe na tela. Às vezes, um simples `ScrollView` pode resolver o problema, mas dependendo do conteúdo da tela é preciso customizar. Outro caso comum é criar um layout específico para tablets de 7" ou 10", com as pastas */res/layout-large* e */res/layout-xlarge*.

30.3 Proporção da tela (aspect ratio)

O termo **aspect ratio** representa a proporção entre a altura e a largura da tela. No Android, essas telas podem ser agrupadas por **aspect ratio** como **long** ou **notlong**.

Uma tela é definida como normal se tiver a resolução HVGA de 320x480px. Neste caso, ela é caracterizada como **notlong**. Saiba que a tela HVGA é definida como a referência padrão para a comparação com outros tamanhos de tela.

Outro grupo que temos são os dispositivos identificados com uma tela do tipo **long**. Para ser definida como **long** uma tela, deve ser considerada comprida quando está na vertical e larga quando está na horizontal. Esse é o caso não apenas dos smartphones mais modernos, que têm uma tela bem comprida, como também dos tablets.

Nota: ao definir o **aspect ratio** da tela, é comum vermos as notações que esta tela é de 3:2 ou 4:3. Por exemplo, uma tela HVGA de 320x480px tem a proporção de 3:2, enquanto uma tela WVGA de 480x800px tem uma proporção de 4:3.

Eu particularmente nunca customizei layouts ou imagens pelo **aspect ratio**, portanto, fique tranquilo, pois na maioria das vezes o que interessa para o desenvolvedor é saber se o dispositivo é um smartphone, tablet de 7" ou tablet de 10".

30.4 Resolução e densidade da tela

Densidade da tela é a relação entre a quantidade de pixels existentes na tela com a sua largura e altura. Por definição, quanto maior a densidade, maior será a quantidade de pixels espalhados pela tela.

No Android, temos aparelhos com densidade baixa (ldpi), densidade normal (mdpi), densidade alta (hdpi), densidade extra alta (xhdpi), densidade extra extra alta (xxhdpi), densidade extra extra extra alta (xxxhdpi); e acho que por enquanto é isso :-).

Podemos dizer que, até o Android 1.5, existiam apenas smartphones de densidade média mdpi, por exemplo, o primeiro HTC G1, HTC Hero e o Motorola DEXT. Mas os smartphones seguintes, como o Nexus One, Sony Ericsson XPeria X10, Motorola Milestone, Samsung Galaxy S, foram lançados geralmente com uma tela física maior (WGA de 480x800px), uma densidade alta hdpi, de forma que são capazes de exibir uma quantidade maior de pixels na tela.

Apenas para conhecimento, na época em que este livro estava sendo escrito, dois dos mais modernos smartphones do Google eram o Nexus 5 e Nexus 6. O Nexus 5 tem uma tela de 1080x1920px com densidade extra extra alta xxhdpi. O Nexus 6 tem uma tela de 1440x2560px com densidade extra extra extra alta xxxhdpi.

Logo a seguir, vamos estudar mais detalhes sobre o conceito de densidade e a diferença entre ldpi, mdpi, hdpi, xhdpi, xxxhdpi etc.

Atenção: o termo “densidade” refere-se à quantidade de pixels disponíveis na tela e não ao tamanho físico da mesma. Podem existir casos de aparelhos terem o mesmo tamanho físico, mas com densidades diferentes.

30.5 O problema de utilizar pixels

Até o Android 1.5, somente existiam celulares Android com o mesmo tamanho de tela, e os desenvolvedores não tinham a preocupação de criar aplicações que funcionassem de forma consistente em telas com diferentes tamanhos e resoluções. Esses celulares com Android 1.5 tinham uma tela HVGA de 320x480px, e alguns dos mais conhecidos na época eram HTC G1, HTC Magic, HTC Hero, Motorola Dext etc.

Com o tempo, foram surgindo dispositivos com telas maiores e com melhor resolução, por isso a partir do Android 1.6 foram criados recursos para auxiliar o desenvolvedor a organizar o aplicativo para que ele funcione perfeitamente em diversos tamanhos de telas.

Para você entender como isso funciona na prática, vamos criar um novo projeto com o nome **Demo-DensityIndependentPixels**. No projeto, vamos criar um **shape**, que é um arquivo XML que representa uma imagem retangular ou oval para geralmente ser utilizada de fundo para alguma view. Eu poderia mostrar os conceitos com qualquer outra view, mas vou mostrar utilizando **shapes** apenas para você aprender mais um recurso do Android. Um **shape** é utilizado como se fosse uma imagem, portanto, crie-o na pasta `/res/drawable`.

 `/res/drawable/oval.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android" android:shape="oval">
    <solid android:color="#cccccc" />
</shape>
```

Feito isso, adicione no arquivo `activity_main.xml` duas imagens, cujo fundo é definido por este **shape**. A primeira imagem tem 100dp de largura e altura, e a segunda tem 100px.

 `/res/layout/activity_main.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout . . . android:orientation="vertical">
    <ImageView
        android:layout_width="100dp" android:layout_height="100dp"
        android:src="@drawable/oval" />
    <ImageView
        android:layout_width="100px" android:layout_height="100px"
        android:layout_marginTop="16dp"
        android:src="@drawable/oval" />
</LinearLayout>
```

No layout temos duas imagens, uma abaixo da outra. A primeira definiu o tamanho em **dp** (density-independent pixels), e a segunda definiu o tamanho em pixels.

A figura 30.1 mostra a pré-visualização desse layout no editor do Android Studio com a opção **Preview All Screen Sizes**. Observe que a primeira bolinha (desenhada acima) está correta. Porém, a segunda bolinha está errada, e o seu tamanho sempre vai variar dependendo da resolução e densidade da tela do dispositivo.

Esse problema ocorreu por que utilizamos a notação **px** (pixel) na segunda imagem. Para solucioná-lo é simples, basta seguir esta regra: “Nunca utilize a notação **px**, sempre utilize a notação **dp**”. Portanto, ao configurar ambas as imagens com largura e altura com 100dp, o resultado será o mesmo em todos os dispositivos, independentemente da resolução da tela, conforme mostra a figura 30.2.

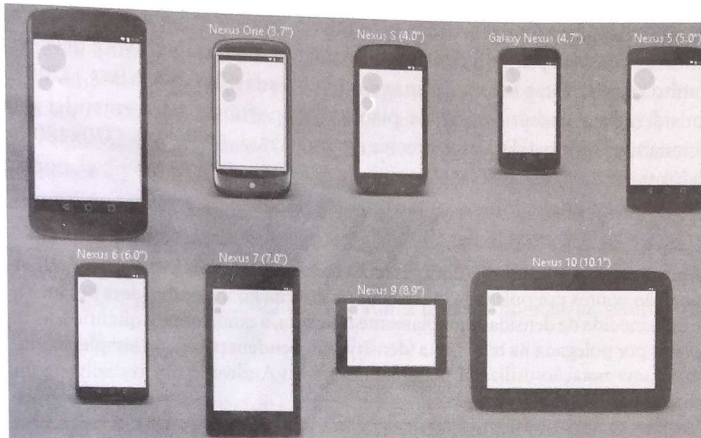


Figura 30.1 – Visualizando o layout em dispositivos com telas diferentes.

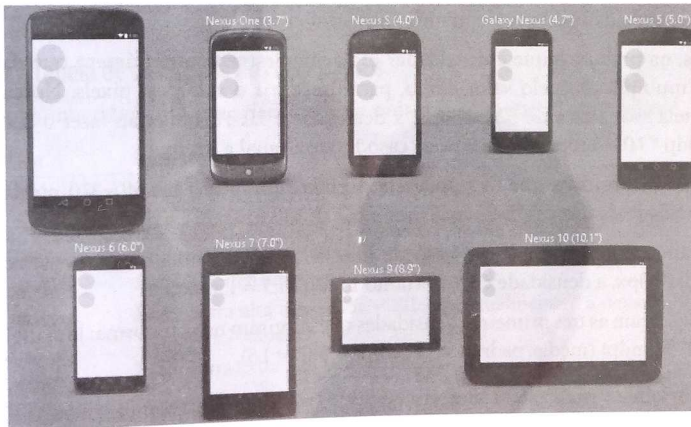


Figura 30.2 – Layout coerente em todos os tamanhos de tela.

30.6 DIP ou DP (density-independent pixel)

A notação dip ou dp é uma unidade de medida de pixel que foi criada para deixar transparente para o desenvolvedor o problema de existir diferentes resoluções e densidades de tela.

Para você entender como a notação `dp` funciona, é preciso tomar como base uma tela `HVGA` de `320x480px`, com densidade normal. Primeiramente, você deve achar estranho eu citar estas telas pequenas e ultrapassadas, mas a tela `HVGA` (`320x480`) é considerada a padrão (base) da plataforma, portanto, para entender como funcionam as outras telas, você precisa entender essa. A tela `HVGA` (`320x480`) tem `160dpi` (Dots per Inch) e, neste caso, a unidade `dp` é equivalente a um pixel, portanto `100dp` é igual a `100px`.

Atenção: não confunda `dpi` (dots per inch) com `dp` (density-independent pixel). A notação pontos por polegada, ou como é conhecida no inglês `dpi` (dots per inch), é uma medida de densidade amplamente utilizada, a qual define a quantidade de pixels por polegada na tela. Já `dp` (density-independent pixel), ou simplesmente `dp`, é uma notação utilizada pela plataforma do Android para trabalhar com diferentes densidades de tela.

O problema é que essa conta pode mudar conforme a tela do dispositivo. Por exemplo, pode ser que `100dp` seja igual a `75px` ou `300px`, tudo depende daquela tal de densidade que comentamos anteriormente.

Mas, na prática, o que é densidade? Ela é simplesmente um número que deve ser multiplicado pelo valor em `dp`, para descobrir o valor em pixels. No caso da tela `HVGA` (`mdpi`) de `320x480px`, a densidade é `1.0`. Portanto, ao fazer a conta $100dp * 1.0 = 100px$, ou seja, neste caso `100dp` é igual a `100px`.

Já em dispositivos que têm uma tela pequena `QVGA` (`ldpi`) de `240x320` pixels, a densidade da tela é `0.75`. Se você fizer a conta, $100dp * 0.75 = 75px$.

Citando outro exemplo, no caso de uma tela alta densidade `WVGA` (`hdpi`) com `480x800px`, a densidade é `1.5`, portanto $100dp * 1.5 = 150px$.

Essas foram as três primeiras densidades que surgiram na plataforma: `ldpi` (baixa = `0.75`), `mdpi` (média/padrão = `1.0`) e `hdpi` (alta = `1.5`).

Mas é claro que você não vai ficar fazendo essas contas, pois justamente por isso foi criada a notação `dp`. Ao utilizá-la, o Android fará a correta conversão de `dp` para pixels utilizando a densidade da tela do dispositivo. Concluindo o raciocínio, como um dispositivo pode ter uma tela com mais ou menos pixels que outras, se utilizarmos valores em pixels, vamos obter resultados diferentes. Por exemplo, em um dispositivo com tela pequena e baixa densidade, `100px` é muita coisa e quase ocupa metade da tela. Mas `100px` nos atuais dispositivos, como Nexus 5 ou Nexus 6, não significa praticamente nada, pois são dispositivos que têm telas com mais de `2.000px`.

Portanto, visualmente 100px em um dispositivo com tela pequena é bastante coisa, porém, em outro com tela grande de alta densidade, 100px quase não seria nada. Veja novamente a figura 30.1 e repare na segunda bolinha que foi desenhada com 100px no Nexus 6. A bolinha mais parece um pequeno pontinho do que uma bola. Isso acontece porque o Nexus 6 contém muitos pixels na tela. A densidade do Nexus 6 é 4.0, portanto, ao desenhar uma bolinha de 100dp, o Android fez a conta: $100dp \cdot 4.0 = 400px$. Por isso, ao utilizar dp, o resultado visual no Nexus 6 ficou bom e coerente com todos os tamanhos de tela (Figura 30.2).

É isso que a notação dp faz! Ela traz o mesmo resultado em todos os tipos de telas. Portanto, lembre-se desta regra: “Nunca utilize a notação px, sempre utilize a notação dp”.

Nota: ao utilizar a notação dp (density-independent pixel), o resultado visual será o mesmo, independentemente da resolução e da densidade da tela. O Android vai usar a densidade da tela para calcular o tamanho correto em pixels.

30.7 Tabela de densidade dos dispositivos

A seguinte tabela mostra a densidade de cada tela suportada pelo Android.

Tela	Densidade
ldpi	Baixa densidade = 0.75.
mdpi	Média densidade = 1.0. É o tamanho padrão para as comparações.
hdpi	Alta densidade = 1.5.
xhdpi	Extra alta densidade = 2.0.
xxhdpi	Extra extra alta densidade = 3.0. Esta atualmente é a densidade dos Nexus 5 e Nexus 6.
xxxhdpi	Extra extra extra alta densidade = 4.0.

Baseado nessa tabela, podemos ver que, quanto maior a densidade da tela, maior é a resolução da mesma, ou seja, maior é a quantidade de pixels que o dispositivo consegue mostrar na tela. Para ter ideia, o Nexus 5 tem uma tela de 5” com 1080x1920px e densidade xxhdpi (3.0). Portanto, ao fazer a conta do exemplo anterior no Nexus 5, fica assim: $100dp \cdot 3.0 = 300px$. Já o Nexus 6 tem uma tela de 6” com 1440x2560px e densidade xxxhdpi (4.0), conhecida como 560dpi. Portanto a mesma conta no Nexus 6 fica: $100dp \cdot 4.0 = 400px$.

30.8 Customizando as imagens conforme a densidade da tela

Agora que já conhecemos o conceito de densidade, vamos explicar como utilizar corretamente a pasta de imagens */res/drawable*.

Você deve ter percebido que esta pasta tem diversas variações conforme a densidade, por exemplo, */res/drawable-ldpi*, */res/drawable-mdpi*, */res/drawable-hdpi*, */res/drawable-xhdpi*, e */res/drawable-xxhdpi*. Para o que serve cada uma dessas pastas acredito que você também saiba, mas talvez a dúvida seja: como devo utilizá-las na prática?

Bom, é assim que eu explico para os designers da minha empresa, então talvez você possa fazer o mesmo.

O tamanho padrão/base de tela é o HVGA com 320x480px, caracterizado como *mdpi* (densidade 1.0). Esta é a pasta */res/drawable-mdpi*. Neste caso, digamos que temos uma imagem de banner que deve ficar no topo da aplicação, portanto para esta tela ela deve ter 320px de largura com alguma altura qualquer, por exemplo, 320x40px. Essa imagem deve ser colocada na pasta */res/drawable-mdpi*.

Ao executar a aplicação em celulares com uma tela de tamanho normal e média densidade, o exemplo vai funcionar perfeitamente. Mas, ao executar a mesma aplicação em um celular de alta densidade (*hdpi*), o Android vai automaticamente procurar as imagens na pasta */res/drawable-hdpi* ou superior. Neste caso, se a imagem não for encontrada nesta pasta, a pasta padrão */res/drawable-mdpi* será utilizada. Mas o problema é que neste caso o Android vai redimensionar a imagem em tempo de execução para se ajustar ao tamanho de tela. O resultado é que a imagem vai se distorcer.

A regra é simples: como em celulares de alta densidade temos mais pixels na tela, consequentemente podemos exibir imagens maiores com mais pixels para obter uma melhor definição e qualidade. Mas como converter a imagem? Qual escala utilizar? Nesses casos, basta multiplicar o tamanho da imagem normal pela densidade desejada, que é a tabela que vimos anteriormente. Por exemplo, para telas *hdpi* (densidade 1.5), basta multiplicar a imagem por 1.5. Portanto, 320x40px multiplicado por 1.5 resulta em 480x60px, e esse é o tamanho da mesma figura que deve ficar na pasta */res/drawable-hdpi*.

Nota: você já parou para ver o tamanho dos ícones de um projeto Android? O ícone da pasta *mipmap-mdpi* tem 48x48px, mas o da pasta *mipmap-hdpi* tem 72x72px. Note que $48 * 1.5 = 72$, onde 1.5 é a densidade de uma tela *hdpi*. Seguindo o mesmo raciocínio, o ícone na pasta *mipmap-xxhdpi* tem 144x144px, pois $48 * 3.0 = 144$, onde 3.0 é a densidade de uma tela *xxhdpi*.

E para finalizar agora vai a dica sobre como trabalhar com imagens.

Se você não informar uma imagem correta para cada densidade em tempo de compilação, ou seja, se não adicionar a imagem no projeto em cada pasta */res/drawable*, o Android vai redimensionar a imagem em tempo de execução. Para isso, ele utiliza a densidade da tela.

A solução para a maioria dos casos é colocar imagens de alta resolução no projeto na pasta */res/drawable-xxhdpi* e deixar o Android converter para baixo em telas menores.

E como os designers devem criar as figuras? Precisa criar as figuras para todos esses tamanhos?

Como eu disse, costumo utilizar um tamanho único, baseado em telas grandes, e insiro as figuras na pasta */res/drawable-xxhdpi*. Eu falo assim para o designer: “Cara, cria as figuras baseados em uma tela de 1080x1920px, que é a do Nexus 5, e deixa o resto comigo”. Nesse caso o Android vai redimensionar a imagem para baixo nos dispositivos com menor densidade e tudo vai funcionar. Eu costumo fazer isso em meus projetos e nunca tive problemas.

Segundo a documentação do Android, as imagens para a densidade *xxhdpi* (3.0) já têm muita definição, então você não precisa utilizar a densidade *xxxhdpi* (4.0) se não quiser, o que reduz o tamanho do aplicativo final.

O Google só recomenda que o ícone do aplicativo seja criado em todas as densidades, pois ao ser redimensionado o ícone apresenta problemas. Portanto, inclusive o ícone de densidade *xxxhdpi* (4.0) precisa ser informado, que é aquele arquivo *ic_launcher.png*.

Nota: se você tem uma imagem e por algum motivo deseja que ela não seja escalada, utilize o qualificador *nodpi*, criando uma pasta */res/drawable-nodpi*.

30.9 Trabalhando com a unidade dp no Canvas

No capítulo 7, sobre a classe *View*, aprendemos a desenhar manualmente no canvas para criar views customizadas. Apenas para lembrar, tudo consiste em criar uma subclasse de *View* e sobrescrever o método *onDraw(canvas)*.

O trecho de código a seguir mostra como desenhar um quadrado de 100x100px.

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    canvas.drawRect(0, 0, 100, 100, paint);
}
```


Opa, mas muita calma nesta hora! Acabamos de utilizar pixels e violar a regra mais importante que é: “Nunca utilize a notação px, sempre utilize a notação dp”.

Ao utilizar a notação dp no código do layout XML, o Android faz a correta conversão de dp para pixels, baseada na densidade da tela do dispositivo. Mas agora no código precisamos recuperar a densidade da tela pela API e fazer o cálculo.

Felizmente isso é simples, basta adicionar um método como este na sua subclasse de View:

```
public float toPixels(float dip) {  
    Resources r = getResources();  
    float densidade = r.getDisplayMetrics().density; // Densidade da tela  
    int px = (int) (dip * densidade + 0.5f); // Converte dp para pixels  
    return px;  
}
```

Com esse método, podemos desenhar o quadrado da maneira correta, que é sempre utilizando valores em dp. Nesse caso, o valor 100dp será convertido para pixels conforme a densidade da tela do dispositivo.

```
@Override  
protected void onDraw(Canvas canvas) {  
    super.onDraw(canvas);  
    canvas.drawRect(0, 0, toPixels(100), toPixels(100), paint);  
}
```

É importante você entender esse conceito para criar views customizadas que funcionem corretamente em todos os tamanhos de telas. Por exemplo, é comum utilizar views customizadas com canvas para desenhar um gráfico de linha, barra ou pizza. Nesses casos, sempre faça essa conversão no código.

Lembre-se de que o segredo do cálculo é recuperar a densidade do aparelho com este método:

```
float densidade = r.getDisplayMetrics().density;
```

Recomendo que você crie um projeto de exemplo qualquer e mostre o valor dessa variável *densidade* em vários emuladores diferentes para fortalecer o conceito. Lembre-se de que a densidade da tela vai retornar sempre valores conforme a tabela que vimos no tópico **“Tabela de densidade dos dispositivos”**.

30.10 Tamanho da tela em dp

Muitas vezes no desenvolvimento para Android é comum encontrar definições de tamanhos de tela em dp. Por exemplo, a documentação oficial do Android diz que 320dp é o tamanho mínimo de uma tela de smartphone, 600dp é o tamanho mínimo de um tablet de 7", e 720dp é o tamanho de um tablet de 10".

Mas como você sabe o tamanho da tela em dp?

Basicamente, o cálculo para descobrir o tamanho da tela em dp é dividir o tamanho da tela em pixels pela densidade. Exemplos:

- A tela HVGA (320x480px) mdpi tem densidade igual a 1, então essa é uma tela de 320x480dp.
- A tela WVGA (480x800px) hdpi tem densidade igual a 1.5, então essa é uma tela 320x533dp.
- O Nexus 5 tem uma tela de (1080x1920px) xxhdpi com densidade 3.0, então essa é uma tela 360x640dp.
- O Nexus 6 tem uma tela de (1440x2560px) xxxhdpi com densidade 4.0, então essa é uma tela 360x640dp.

30.11 Dimensões (dimen)

Para fechar o assunto sobre conversão de dp para pixels com chave de ouro, vamos falar sobre os famosos `dimens`, que são constantes que ficam no arquivo `/res/values/dimens.xml`.

A vantagem de utilizar constantes é porque podemos reaproveitá-las em um único lugar. No caso dos `dimens` ainda existe outra vantagem, pois um valor de dimensão (`dimen`) automaticamente converte o valor de dp para pixels. Por exemplo, digamos que temos as seguintes constantes para definir a largura e altura de um quadrado, o qual será desenhado pela API utilizando `canvas`.

```
 /res/values/dimens.xml

public class ConverterPixelDP1Activity extends Activity {
<resources>
    <dimen name="quadrado_width">100dp</dimen>
    <dimen name="quadrado_height">100dp</dimen>
</resources>
```

Ao ler cada dimensão no código, o valor é retornado automaticamente em pixels.

```
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    // Lê a dimensão. O valor será retornado em pixels conforme a densidade da tela
    float larguraPx = getContext().getResources().getDimension(R.dimen.quadrado_width);
    float alturaPx = getContext().getResources().getDimension(R.dimen.quadrado_height);
    // Desenha o quadrado
    canvas.drawRect(0, 0, larguraPx, alturaPx, paint);
}
```

Pronto! Além de utilizar constantes, que é uma boa prática de programação, não precisamos fazer nenhum cálculo, pois um `dimen` já retorna o valor em pixels correto.

Dimensões, ou apenas `dimens` como são popularmente conhecidos, também são muito utilizados em arquivos de layout XML para definir as configurações de espaçamento e margens. Por exemplo, atualmente o wizard do Android Studio ao gerar um arquivo XML cria as configurações de padding (espaçamento) da seguinte forma:

```
 /res/layout/fragment_main.xml

<RelativeLayout . . .
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin" >
    <TextView . . . />
</RelativeLayout>
```

Essas dimensões são referenciadas com a sintaxe `@dimen/nome` e ficam no arquivo `/res/values/dimens.xml`.

```
 /res/values/dimens.xml

<resources>
    <!-- Segundo a UI Guideline do Android, 16dp é o recomendado para margens. -->
    <dimen name="activity_horizontal_margin">16dp</dimen>
    <dimen name="activity_vertical_margin">16dp</dimen>
</resources>
```

As dimensões são utilizadas para especificar espaçamentos, tamanhos de fonte, tamanhos de view etc. e, dependendo do tamanho da tela, elas podem ser customizadas. Por exemplo, a notação `/res/values-w820dp` sobrescreve a dimensão para tablets de 7" ou 10" caso estejam na horizontal. O tablet de 7" na horizontal tem uma tela de aproximadamente 960dp e o tablet de 10" tem uma tela de 1280dp.

Baseado nessas informações, a notação */res/values-w820dp* indica que esta pasta será utilizada caso a largura da tela tenha pelo menos 820dp. O "w" neste caso significa width (largura).

 */res/values-w820dp/dimens.xml*

```
<resources>
  <!-- Valor para tablets de 7" ou 10" na horizontal. -->
  <dimen name="activity_horizontal_margin">64dp</dimen>
</resources>
```

Criar dimensões também é muito comum para manter aplicativos compatíveis com smartphones e tablets. Por exemplo, podemos ter um `TextView` com o tamanho de fonte 14sp desta forma:

```
<TextView ... android:text="..." android:textSize="@dimen/text_size" />
```

A notação `@dimen/text_size` vai ler a dimensão definida no arquivo */res/values/dimens.xml*.

 */res/values/dimens.xml*

```
<resources>
  <dimen name="text_size">14sp</dimen>
</resources>
```

Mas para os tablets de 10 polegadas podemos sobrescrever esta dimensão e aumentar a fonte do texto, basta criar o seguinte arquivo:

 */res/values-xlarge/dimens.xml*

```
<resources>
  <dimen name="text_size">40sp</dimen>
</resources>
```

Ou podemos utilizar a notação `sw` de `smallest width` (menor largura), a qual é compatível com o Android 3.2 ou superior e será explicada no próximo tópico.

 */res/values-sw720dp/dimens.xml*

```
<resources>
  <dimen name="text_size">40sp</dimen>
</resources>
```

Nota: sempre utilize a notação `sp` para tamanho de fontes, da mesma forma que `dp` para tamanho e espaçamento de views.

30.12 Qualificadores de recursos para tablets

Ao desenvolver aplicativos para Android, é preciso customizar a interface para os tablets. Utilizar o mesmo layout do smartphone e apenas esticar ou aumentar o conteúdo pode não ser o ideal, pois a experiência ao utilizar o aplicativo não é otimizada para tablets.

Um exemplo clássico de layout para tablets é a tela dividida em duas partes, para aproveitar melhor o espaço disponível na tela. Porém, no capítulo 8, sobre fragments, também estudamos vários tipos de layouts que podem ser utilizados.

Para criar um layout específico para tablets, podemos usar as seguintes pastas:

Pasta	Descrição
<i>/res/layout-large</i>	Para tablets de 7" (large = grande).
<i>/res/layout-xlarge</i>	Para tablets de 10" (xlarge = extragrande).

Mas a partir do Android 3.2 foram criados outros qualificadores, que comparam o tamanho da tela em dp (largura ou altura) para identificar se o layout é para smartphone ou tablets.

Identificador	Descrição
<i>height dp</i>	Altura disponível – Altura mínima da tela, mas pode mudar conforme a orientação.
<i>width dp</i>	Largura disponível – Idem ao anterior, mas compara a largura da tela.
<i>smallest width dp</i>	Menor largura da tela, sem levar em consideração a orientação.

Utilizando esses qualificadores, é possível criar pastas com a notação */res/layout-h600dp* para definir a altura desejada, */res/layout-w600dp* para definir a largura e */res/layout-sw600dp* para definir a largura mínima da tela, independentemente se estiver na vertical ou horizontal.

Por exemplo, se o layout exige que a menor dimensão da tela tenha pelo menos 600dp, sempre (independentemente da orientação), podemos usar o qualificador */res/layout-sw600dp/* para criar os recursos de layout.

Desses identificadores, o *smallest width dp* (menor largura) é o mais utilizado e substitui as notações *large* e *xlarge*. O indicador *smallest width dp* é baseado na largura da tela, pois a largura é frequentemente utilizada como base para criação de um layout. Na altura (vertical) geralmente não temos muitos problemas para criar layouts, pois é possível criar uma tela com rolagem. Mas a largura é um fator decisivo para decidir a quantidade de conteúdo que cabe na tela. Muitas vezes, dependendo desse conteúdo, podemos decidir entre criar layouts diferentes no

aplicativo para smartphones e para tablets. Por exemplo, se a tela é bem larga, podemos até dividi-la em duas partes (esquerda e direita).

E qual a largura dos dispositivos? Como vou saber qual largura utilizar? Para isso existe esta tabelinha da documentação, que mostra a largura mínima em dp para cada tipo de tela.

Largura	Descrição
320dp	Esta é largura mínima dos smartphones (240x320ldpi, 320x480mdpi, 480x800hdpi etc.).
600dp	Tablet de 7" (600x1024px).
720dp	Tablet de 10" (720x1280, 800x1280px).

Baseado nesses seletores por tamanho da largura da tela, fica fácil criar layouts específicos para smartphones ou tablets, como por exemplo:

Pasta	Descrição
<i>/res/layout</i>	Pasta de layout para smartphones.
<i>/res/layout-sw600dp</i>	Pasta de layout para tablets de 7" e 10".

Caso você precise criar layouts diferenciados para tablets de 7" e 10", basta utilizar estas pastas:

Pasta	Descrição
<i>/res/layout</i>	Pasta de layout para smartphones.
<i>/res/layout-sw600dp</i>	Pasta de layout para tablets de 7".
<i>/res/layout-sw720dp</i>	Pasta de layout para tablets de 10".

Basicamente, essas duas pastas indicam o tamanho da tela na horizontal; a notação *sw* é de *smallest width* (menor largura). Neste caso, os tablets de 7" têm pelo menos 600dp de largura e os de 10" têm pelo menos 720dp de largura.

Mas, como eu já disse antes, ainda sou adepto de utilizar principalmente a pasta */res/layout-xlarge* para tablets de 10", pois essa notação funciona desde o Android 3.0 e não apenas a partir do Android 3.2.

30.13 Links úteis

Este capítulo explicou os principais conceitos sobre como desenvolver aplicativos para vários tamanhos de telas, com resoluções e densidades diferentes. Mas o tema é complexo e por isso recomendo que você complemente a leitura com textos da documentação oficial e blog do Google.

Como eu disse na introdução, este capítulo iria lhe tirar muitas dúvidas, ou deixá-lo com muitas dúvidas. Portanto, caso você não tenha entendido muito bem algo, não tem problema, pois o assunto é complicado mesmo. Continue lendo o livro e mais tarde leia novamente esse assunto se precisar. Eu espero que este livro sirva de guia não para você somente aprender o básico sobre o Android, mas para ser uma fonte de consulta quando quiser relembrar algum conceito.

Mais do que nunca, é importante complementar a leitura com a documentação oficial.

- **Providing Resources**

<http://developer.android.com/guide/topics/resources/providing-resources.html>

- **Supporting Multiple Screens**

http://developer.android.com/guide/practices/screens_support.html

- **Supporting Tablets and Handsets**

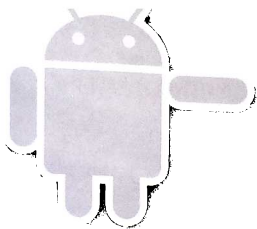
<http://developer.android.com/guide/practices/tablets-and-handsets.html>

- **New Tools for Managing Screen Sizes**

<http://android-developers.blogspot.com.br/2011/07/new-tools-for-managing-screen-sizes.html>

- **Getting Your Apps Ready for Nexus 6 and Nexus 9**

<http://android-developers.blogspot.com.br/2014/10/getting-your-apps-ready-for-nexus-6-and.html>



CAPÍTULO 31

Threads avançado – AsyncTask e Loader

Neste capítulo, vamos estudar alguns exemplos mais avançados da classe `AsyncTask`, assim como a nova API criada no Android 3.0, chamada de `Loader`.

O objetivo deste capítulo é explicar situações e erros comuns que geralmente costumam dar trabalho para os desenvolvedores Android. O assunto deste tópico é avançado, portanto leia quando achar necessário.

31.1 O problema com o `ProgressDialog`

Nesta altura do campeonato, você já sabe a importância de utilizar threads e já conhece a classe `AsyncTask`, a qual facilita a utilização de threads no Android. Portanto, vamos direto ao assunto e estudar alguns assuntos avançados e erros comuns enfrentados por desenvolvedores.

Apenas para lembrar, eu particularmente sempre utilizo em meus projetos uma pequena biblioteca que encapsula a classe `AsyncTask`, pois, conforme expliquei no capítulo 17, sobre web services, e vimos no projeto dos carros, tivemos várias vantagens encapsulando o código da `AsyncTask`. Nestes próximos exemplos, utilizarei diretamente a classe `AsyncTask` apenas para demonstrar alguns problemas clássicos e assuntos avançados sobre o ciclo de vida da aplicação.

Abra o projeto de exemplo deste capítulo e procure a activity que demonstra como fazer o download de uma imagem. O exemplo de download é o mesmo que fizemos no capítulo 10, sobre threads e handler, porém, em vez de utilizar um `ProgressBar` para fazer a animação, foi usado um alerta com o `ProgressDialog`. Justamente isso nos trará um problema.



DownloadImagemAsyncTaskActivity.java

```
public class DownloadImagemAsyncTaskActivity extends AppCompatActivity {
    private static final String URL = "http://livroandroid.com.br/imgs/livro_android.png";
    private ProgressDialog progress;
    private ImageView imgView;
    private Bitmap bitmap;
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_download_imagem);
        imgView = (ImageView) findViewById(R.id.img);
        if(icle != null) {
            // Recupera o estado
            bitmap = icle.getParcelable("img");
        }
        if(bitmap == null) {
            // Faz o download
            downloadImagem(URL);
        } else {
            // Atualiza a imagem se recuperou o estado
            imgView.setImageBitmap(bitmap);
        }
    }
    // Faz o download da imagem em uma nova thread
    private void downloadImagem(final String urlImg) {
        // Cria uma AsyncTask
        DownloadTask task = new DownloadTask();
        // Executa a task/thread
        task.execute(URL);
    }
    private class DownloadTask extends AsyncTask<String,Void,Bitmap> {
        @Override
        protected void onPreExecute() {
            super.onPreExecute();
            // Mostra o ProgressDialog antes de iniciar a task
            progress = ProgressDialog.show(getContext(),"Aguarde",
                "Fazendo o download... Gire a tela durante o download para fazer BOOM!");
        }
        @Override
        protected Bitmap doInBackground(String... params) {
            // Faz o download da imagem
            try {
```

```

        bitmap = Download.downloadBitmap(URL);
    } catch (Exception e) {
        Log.e("livroandroid", e.getMessage(), e);
    }
    return bitmap;
}

protected void onPostExecute(Bitmap bitmap) {
    if(bitmap != null) {
        Toast.makeText(getBaseContext(), "OK", Toast.LENGTH_SHORT).show();
        // Esconde o progress
        closeProgress();
        // Atualiza a imagem
        imageView.setImageBitmap(bitmap);
    }
}

@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    // Salva o estado da tela
    outState.putParcelable("img", bitmap);
}

private void closeProgress() {
    if(progress != null && progress.isShowing()) {
        progress.dismiss();
        progress = null;
    }
}

public Context getApplicationContext() {
    return this;
}
}

```

O layout desta activity é simples e contém apenas o `ImageView`.

 `/res/layout/activity_download_imagem.xml`

```

<?xml version="1.0" encoding="utf-8"?>
<FrameLayout . . . >
    <ImageView android:id="@+id/img" android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center" android:scaleType="fitCenter" />
</FrameLayout>

```

O resultado deste exemplo pode ser visto na figura 31.1, e a princípio tudo está funcionando bem.



Figura 31.1 – Download da imagem realizado com sucesso.

Esse código mostrou como fazer algumas coisas:

1. O download foi feito em uma thread com a classe `AsyncTask`.
2. Estou utilizando um `ProgressDialog` para mostrar o problema de girar a tela (veja próximo tópico). Na prática, prefiro utilizar o `ProgressBar` conforme mostrado em outros exemplos, mas quero reforçar um problema clássico com o `ProgressDialog`.
3. Se, depois do download, o usuário girar a tela, a activity vai salvar o `Bitmap` no método `onSaveInstanceState(bundle)` e recuperar esse mesmo `Bitmap` depois quando for recriada.
4. Se durante o download o usuário girar a tela, a aplicação vai travar.

Sobre o último item explicado acima, vamos testar para simular o erro. Execute o aplicativo e, enquanto o `ProgressDialog` está aberto, gire a tela. O resultado é um erro conforme a figura 31.2.

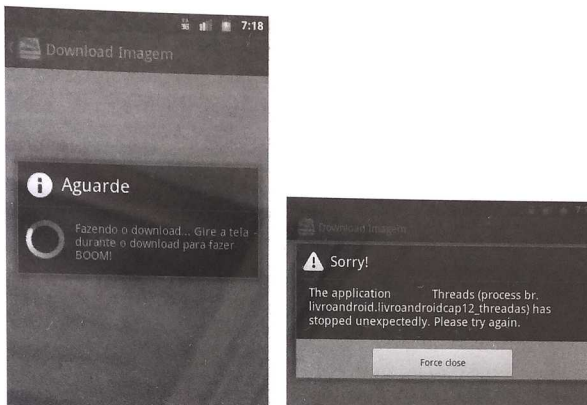


Figura 31.2 – Erro ao girar a tela durante o download.

No LogCat podemos ver a exceção detalhada (stack trace), conforme a figura 31.3.

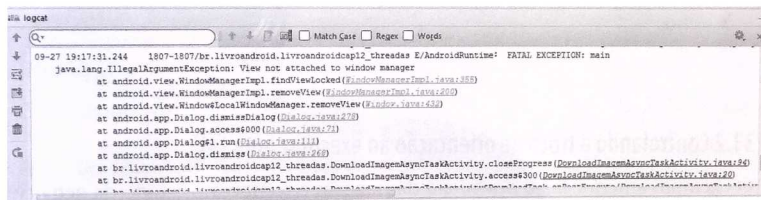


Figura 31.3 – Erro ao girar a tela durante o download.

Esse erro acontece porque a tarefa da AsyncTask continua executando em segundo plano durante a rotação da tela, e quando o download da imagem termina a aplicação tenta fechar o ProgressDialog. Como o ProgressDialog é associado com a activity que o criou, esse erro ocorre justamente porque essa activity foi destruída durante a troca de orientação.

Para solucionar esse problema, temos de fechar o ProgressDialog quando a activity for encerrada, mesmo antes de a AsyncTask terminar, conforme demonstrado a seguir:

 DownloadImagemAsyncTaskActivity.java

```
public class DownloadImagemAsyncTaskActivity extends AppCompatActivity {
    ... // Tudo igual aqui ...
    @Override
    protected void onDestroy() {
```

```
        super.onDestroy();  
        closeProgress();  
    }  
}
```

Note que é o mesmo código mostrado anteriormente, mas adicionei o método `onDestroy()`. Feito isso, a aplicação não vai mais travar, pois o `ProgressDialog` é encerrado junto com a `activity`. Naturalmente, o ideal também seria parar a tarefa da `AsyncTask` chamando o método `cancel(boolean)`.

Porém ainda temos um problema, pois, quando a nova `activity` for criada depois da rotação da tela, o download da imagem vai iniciar novamente. Neste caso precisamos dar um jeito de girar a tela sem interromper o download e reutilizar a primeira thread que está fazendo o download da imagem.

Nota: estou mostrando esses problemas ou situações porque com certeza isso algum dia vai acontecer em algum aplicativo seu, e assim você já estará preparado. Se precisar, releia este capítulo depois, pois estamos dando um passo além do tradicional, que é mostrar como consultar um web service com uma thread ou `AsyncTask`.

31.2 Controlando a troca de orientação ao executar uma task

O exemplo anterior faz o download da imagem e recupera o estado da `activity` caso o usuário gire a tela depois de o download terminar. Mas caso o usuário gire a tela durante o download temos um problema.

Quando o Android recriar a `activity`, do jeito que está o código, no método `onCreate(bundle)` será disparada uma nova `AsyncTask` para fazer o download da imagem, sendo que a primeira thread ficou perdida em memória. Isso significa que se você ficar girando a tela várias vezes o download nunca vai terminar, pois várias threads serão disparadas, mas nenhuma delas vai conseguir atualizar a interface da nova `activity`.

Para solucionar esse problema, vamos utilizar um fragment com o método `setRetainInstance(true)` para deixar a instância do fragment viva, ou seja, reter o fragment em memória. Dessa forma, a `AsyncTask` continuará viva dentro do fragment, e quando ela terminar basta atualizar a interface. Somente a view do fragment será recriada, mas a `AsyncTask` e o download serão mantidos e reutilizados.

O seguinte código mostra a `activity` que apenas vai adicionar o fragment no layout.

 DownloadImagemFragmentActivity.java

```
public class DownloadImagemFragmentActivity extends AppCompatActivity {
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        setContentView(R.layout.activity_download_imagem_fragment);
        if(icle == null) {
            getSupportFragmentManager().beginTransaction().add(R.id.layoutFrag,
                new DownloadImagemFragment()).commit();
        }
    }
}
```

 /res/layout/activity_download_imagem_fragment.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:id="@+id/layoutFrag">
    <!-- fragment aqui -->
</FrameLayout>
```

Como podemos ver, a activity não faz nada, pois a lógica fica toda no fragment.

 DownloadImagemFragment.java

```
public class DownloadImagemFragment extends Fragment {
    private static final String URL = "http://livroandroid.com.br/imgs/livro_android.png";
    private ProgressDialog progress;
    private Bitmap bitmap;
    private ImageView imgView;
    // A task fica como atributo para ficar viva durante a rotação da tela
    private DownloadTask task;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setRetainInstance(true); // Salva o estado do fragment
    }
    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
        @Nullable Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_download_imagem, container, false);
        imgView = (ImageView) view.findViewById(R.id.img);
    }
}
```

```

Log.d("livroandroid", "frag onCreateView()");
if(bitmap == null) {
    // Faz o download
    downloadImagem(URL);
} else {
    // Atualiza a imagem se recuperou o estado
    setBitmap(bitmap);
}
return view;
}

// Faz o download da imagem em uma nova thread
private void downloadImagem(final String urlImg) {
    // O atributo task fica retido em caso de rotação, assim podemos saber
    // se a task já está executando
    if(task == null) {
        Log.d("livroandroid", "> DownloadTask.execute()");
        // Cria uma AsyncTask
        task = new DownloadTask();
        // Executa a task/thread
        task.execute(URL);
    } else {
        Log.d("livroandroid", "DownloadTask já está executando.");
        // Mostra novamente o dialog para acompanhar o download
        showProgress();
    }
}

private class DownloadTask extends AsyncTask<String, Void, Bitmap> {
    @Override
    protected void onPreExecute() {
        Log.d("livroandroid", "task onPreExecute()");
        showProgress();
    }
    @Override
    protected Bitmap doInBackground(String... params) {
        Log.d("livroandroid", "task doInBackground()");
        // Faz o download da imagem
        Bitmap bitmap = null;
        try {
            bitmap = Download.downloadBitmap(URL);
        } catch (Exception e) {
            Log.e("livroandroid", e.getMessage(), e);
        }
        return bitmap;
    }
}

```

```

    }
    @Override
    protected void onCancelled() {
        task = null;
        super.onCancelled();
        Log.d("livroandroid", "task onCancelled()");
    }
    protected void onPostExecute(Bitmap imagem) {
        task = null;
        Log.d("livroandroid", "task onPostExecute()");
        if (imagem != null) {
            setBitmap(imagem);
        }
    }
}

private void setBitmap(Bitmap imagem){
    Log.d("livroandroid", "setBitmap()");
    this.bitmap = imagem;
    // Fecha o progress
    closeProgress();
    // Atualiza a imagem
    imageView.setImageBitmap(imagem);
}

@Override
public void onDetach() {
    super.onDetach();
    Log.d("livroandroid", "frag onDetach()");
    // Fecha o progress antes de desassociar o fragment da activity
    closeProgress();
}

private void showProgress() {
    Log.d("livroandroid", "showProgress()");
    // Mostra o dialog e permite cancelar
    progress = ProgressDialog.show(getActivity(), "Aguarde", "Fazendo o download...");
    progress.setCancelable(true);
    progress.setOnCancelListener(new DialogInterface.OnCancelListener() {
        @Override
        public void onCancel(DialogInterface dialog) {
            // Cancela a AsyncTask
            task.cancel(true);
        }
    });
}
}

```



```

private void closeProgress() {
    Log.d("livroandroid", "closeProgress()");
    if(progress != null && progress.isShowing()) {
        progress.dismiss();
        progress = null;
    }
}
}
}

```

Como podemos ver neste exemplo, o segredo é deixar a `AsyncTask` dentro de um `fragment` que fica retido em memória. Neste caso, se você girar a tela o download vai continuar executando normalmente. O segredo do código é manter o atributo `task` na classe para controlar se a `AsyncTask` já está executando ou não.

Este código mostrou como fazer algumas coisas:

1. O download foi feito em uma thread com a classe `AsyncTask`.
2. O usuário pode girar a tela durante o download sem interromper a task.
3. O estado da tela fica salvo, pois o `fragment` é retido em memória.
4. Foi implementado o listener `DialogInterface.OnCancelListener` no `ProgressDialog` para permitir que ele seja cancelado. Para fechar a janela do `ProgressDialog`, pressione o botão **voltar** do Android. Ao cancelar a janela, a `AsyncTask` também é cancelada chamando o método `task.cancel(true)`.

Nota: quando a `AsyncTask` é cancelada com a chamada do método `cancel(true)`, o método `onPostExecute()` não será chamado, no seu lugar será chamado o método `onCancelled()`. Se em qualquer momento você precisar saber se a tarefa está cancelada, basta chamar o método `isCancelled()`.

Para ajudá-lo a estudar e entender o código, recomendo fazer estes dois testes e verificar os logs do LogCat. O primeiro teste é fazer o download e aguardar até a imagem ser exibida. Neste caso, teremos os seguintes logs:

```

D/livroandroid: frag onCreateView() // fragment criado
D/livroandroid: > DownloadTask.execute() // task iniciada
D/livroandroid: task onPreExecute() // pré-executa na UI Thread
D/livroandroid: showProgress() // mostra o progress dialog
D/livroandroid: task doInBackground() // faz o download em background
D/livroandroid: task onPostExecute() // fim do download na UI Thread
D/livroandroid: setBitmap() // atualiza a imagem
D/livroandroid: closeProgress() // fecha o progress dialog

```

- O segundo teste é iniciar o download e girar a tela antes de o download terminar.
- O resultado dos logs deverá ser assim:

```
D/livroandroid: frag onCreateView() // fragment criado
D/livroandroid: > DownloadTask.execute() // task iniciada
D/livroandroid: task onPreExecute() // pré-executa na UI Thread
D/livroandroid: showProgress() // mostra o progress dialog
D/livroandroid: task doInBackground() // faz o download em background
// Aqui eu girei a tela
D/livroandroid: frag onDetach() // O fragment será desassociado da activity
D/livroandroid: closeProgress() // fecha o progress dialog antes de a activity
// ser destruída
// A activity e o fragment são recriados
D/livroandroid: frag onCreateView() // depois da rotação da tela, vamos recriar
// a view do fragment
D/livroandroid: DownloadTask já está executando. // como o fragment foi retido, vimos
// que o download ainda está executando
D/livroandroid: showProgress() // mostra o progress dialog novamente, pois o
// download ainda está executando
D/livroandroid: task onPostExecute() // fim do download
D/livroandroid: setBitmap() // atualiza a imagem (vai utilizar a nova
// view do fragment)
D/livroandroid: closeProgress() // fecha o progress dialog
```

Neste tópico, você aprendeu a implementar uma AsyncTask corretamente com um Fragment e tratar os problemas de troca de orientação da tela. Isso costuma ser um problema para muitos desenvolvedores Android, mas com este exemplo você terá uma boa base para construir seus aplicativos.

31.3 Executando a AsyncTask de forma serial ou paralela

Quando a AsyncTask foi criada, todas as tarefas/tasks executavam de forma serial (uma após outra) em uma única thread. Internamente a AsyncTask controla essa fila e o pool de threads.

A partir do Android 1.6, esse comportamento foi alterado para permitir que as tasks executassem em paralelo. No caso de um aplicativo para tablet com diversos fragments espalhados pela tela, isso pode ser útil, pois a task de cada fragment pode executar em paralelo sem depender da resposta de outro. Porém, em aplicações nas quais a ordem da execução das tarefas é importante, esse comportamento pode não ser o ideal.

Por isso o Google a partir do Android 3.0 voltou ao comportamento original, e todas as tarefas executam em uma única thread serialmente. Mas pela API podemos escolher se desejamos iniciar a `AsyncTask` de forma serial (padrão) ou paralela.

O seguinte código mostra como executar a tarefa de forma serial (padrão):

```
AsyncTask task = ...;
task.execute(parâmetros aqui);
// Isso é a mesma coisa que...
task.executeOnExecutor(AsyncTask.SERIAL_EXECUTOR, parâmetros aqui);
```

E para executar a tarefa de forma paralela é assim:

```
AsyncTask task = ...;
task.executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR, parâmetros aqui);
```

Podemos ver nestes códigos que é utilizado um `Executor` da linguagem Java. A API de `Executor` do Java foi criada para gerenciar um pool de threads facilitando a implementação de programação concorrente. Caso queira estudar mais detalhes de como isso funciona internamente, pesquise sobre Java `Executor`.

Atenção: o método `executeOnExecutor(executor,params)` foi criado no Android 3.0, portanto só pode ser utilizado em dispositivos com API Level 11 ou superior. Se você executar esse código em versões anteriores do Android, o aplicativo vai lançar uma exceção em tempo de execução. De qualquer forma, é importante entender que no Android 1.6 até o 2.x as tarefas executam em paralelo por padrão, mas no Android 3.0 ou superior as tarefas executam de forma serial por padrão.

31.4 Loader

Antes de você ler sobre loaders, gostaria de ressaltar que este tópico é ligeiramente avançado. Talvez a leitura possa ser um pouco pesada, portanto leia quando achar necessário.

Loaders foram criados no Android 3.0 como a forma oficial de executar tarefas assíncronas no Android. A interface `Loader` faz parte do pacote `android.content.Loader`, mas para funcionar em todas as versões do Android podemos utilizar a interface `android.support.v4.content.Loader` da biblioteca de compatibilidade v4.

Um loader pode ser utilizado dentro de uma `activity` ou `fragment` e é utilizado para executar tarefas assíncronas. Uma das suas principais funcionalidades é monitorar a fonte de dados para que, em caso de mudanças, a interface da tela possa ser atualizada rapidamente. Essa funcionalidade não é muito utilizada em

aplicativos que utilizam web services, pois para saber se a fonte de dados mudou precisamos fazer requisições no servidor web. Mas existem casos em que isso pode ser interessante. Por exemplo, se o seu aplicativo mostra a lista de contatos da agenda, sempre que o usuário alterar as informações de algum contato, o aplicativo pode ser avisado, para atualizar a lista com as informações corretas.

Mas talvez a principal vantagem do loader é que ele sobrevive durante uma mudança de configuração do sistema, como a troca de orientação. Como vimos anteriormente, podemos utilizar um fragment e retê-lo em memória para solucionar esse problema com a classe `AsyncTask`, mas um loader já faz esse gerenciamento de forma automática. Caso o usuário gire a tela, por exemplo, o sistema vai simplesmente reconectar a tarefa no loader que já existe.

Eu particularmente me vejo bem satisfeito com a solução mostrada anteriormente, com um fragment retido em memória. No entanto, utilizar loaders é a maneira recomendada pelo Google para executar tarefas assíncronas, então vamos lá!

Para utilizar um loader, precisamos implementar a interface `Loader`. Um loader é responsável por executar determinada tarefa em segundo plano e retornar o resultado. O melhor de tudo é que o loader também pode ser desvinculado da `activity` ou fragment.

Como o loader é uma interface, existem duas implementações nativas padrão da plataforma, as classes `CursorLoader` e `AsyncTaskLoader`. A classe `CursorLoader` faz a leitura de um content provider, que é utilizado para ler a agenda de contatos, dentre outras coisas. Já a classe `AsyncTaskLoader` implementa a interface `Loader` e utiliza uma `AsyncTask` internamente para executar a tarefa em background, portanto veja a importância de termos estudado a classe `AsyncTask`. O loader simplesmente delega o trabalho para a `AsyncTask`!

Dica: uma das principais funcionalidades de um loader é monitorar a fonte de dados para que, em caso de mudanças, a interface da tela possa ser atualizada rapidamente. No capítulo 32, sobre provedores de conteúdo, vamos utilizar a classe `CursorLoader` para ler os contatos da agenda e ver essas vantagens na prática. Neste capítulo, vamos estudar os conceitos sobre a API de Loaders.

Para executar um loader, é utilizada a classe `LoaderManager`, a qual é uma classe abstrata que está associada com a `activity` ou o fragment que está executando. O código a seguir mostra como iniciar um loader:

```
getLoaderManager().initLoader(id, args, callback);
```

O método `initLoader(id, args, callback)` contém os seguintes parâmetros:

int id

Identificador único para o loader. Se existe apenas um loader, o identificador pode ser 0 (zero).

Bundle args

Argumentos para construir o loader ou null se não for necessário.

LoaderCallbacks<D> callback

Implementação da interface `LoaderManager.LoaderCallbacks` que será responsável por criar o loader e receber o resultado retornado pelo loader. É como se esta interface gerenciasse o ciclo de vida do loader, sendo responsável por criá-lo, receber todos os eventos e depois receber o resultado.

Para entendermos isso na prática, podemos criar um loader que faz o download de uma imagem. Veja que a classe `ImageLoader` herda de `AsyncTaskLoader<Bitmap>` e, como o tipo `Bitmap` foi utilizado no argumento genérico, o método `loadInBackground()` deve retornar um `Bitmap`.



`ImageLoader.java`

```
public class ImageLoader extends AsyncTaskLoader<Bitmap> {
    private static final String URL = "http://livroandroid.com.br/imgs/livro_android.png";
    public ImageLoader(Context context) {
        super(context);
    }
    @Override
    protected void onStartLoading() {
        super.onStartLoading();
        Log.d("livroandroid", "loader onStartLoading()");
        // Precisa chamar o forceLoad() para executar o loader "loadInBackground"
        forceLoad();
    }
    @Override
    public Bitmap loadInBackground() {
        Log.d("livroandroid", "loader loadInBackground()");
        // Faz o download da imagem
        Bitmap bitmap = null;
        try {
            bitmap = Download.downloadBitmap(URL);
        } catch (Exception e) {
            Log.e("livroandroid", e.getMessage(), e);
        }
    }
}
```

```

        return bitmap;
    }
}

```

Ao chamar o método `getLoaderManager().initLoader(id,args,callback)` para iniciar o loader, o método `onStartLoading()` é chamado. Sua responsabilidade é decidir se o loader é iniciado ou não. Podemos dizer que no método `onStartLoading()` você deve dar o comando para iniciar a tarefa de verdade, e isso é feito chamando o método `forceLoad()`.

O método `loadInBackground()` é executado em segundo plano por uma thread, e é aqui que você deve fazer a busca no web service ou banco de dados.

Nota: observe como a classe `ImageLoader` é independente da activity ou fragment. A única tarefa que ela faz é buscar um `Bitmap` de um endereço URL. Então podemos dizer que o `Loader` é somente a classe responsável por executar determinada tarefa e retornar um objeto como resultado.

Mas o loader sozinho não faz nada, pois alguém precisa chamá-lo. Por isso utilizamos a classe `LoaderManager` e o método `getLoaderManager().initLoader(id,args,callback)`. Neste caso, o terceiro parâmetro `callback` deve implementar a interface `LoaderManager.LoaderCallbacks` que contém estes três métodos:

```
Loader<D> onCreateLoader(int id, android.os.Bundle bundle);
```

Método que retorna alguma implementação da interface `Loader`, como `CursorLoader` ou `AsyncTaskLoader`.

```
void onLoadFinished(Loader<D> loader, D d);
```

Quando o loader terminar de carregar os dados, este método é chamado na UI Thread para atualizar a view.

```
void onLoaderReset(Loader<D> loader);
```

Este método é chamado caso o loader seja limpo (feito o reset). Para limpar um loader, basta chamar o método `reset()`, o que significa que o loader deve eliminar os dados para economizar recursos e memória.

Então podemos dizer que um template básico para executar um loader seria como o código demonstrado a seguir. Veja que a implementação da interface `LoaderManager.LoaderCallbacks` é o segredo da chamada. Este callback deve criar o loader e depois receber a resposta da execução no método `onLoadFinished()`.

```

public class TemplateStrobesExecutorLoader extends Fragment {
    @Override
    public View onCreateView(LayoutInflater inflater, (Nullable ViewGroup) container,
        (Nullable Bundle) savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_download_image, container, false);
        // Inicia o loader ou reconecta a um já existente
        getLoaderManager().initLoader(0, null, new LoaderCallbacks());
        return view;
    }

    private class LoaderCallbacks implements LoaderManager.LoaderCallbacks<Bitmap> {
        @Override
        public Loader<Bitmap> onCreateLoader(int id, Bundle args) {
            // Método chamado para criar o loader quando necessário
            // Deve criar e retornar a classe do loader
            return new ImageLoader(getActivity());
        }

        @Override
        public void onLoadFinished(Loader<Bitmap> loader, Bitmap data) {
            // Chamado para entregar a resposta do loader e atualizar a view
            // Executa na UI Thread
            // Neste caso do download, recebemos o Bitmap aqui
        }

        @Override
        public void onLoaderReset(Loader<Bitmap> loader) {
            // Chamado caso o loader tenha sido limpo (feito o reset)
        }
    }
}

```

Depois desse template básico para você entender a ideia, vamos mostrar um exemplo completo. O objetivo será fazer o download da imagem novamente. Nesse caso, vamos utilizar a mesma activity do exemplo anterior, pois vamos apenas alterar a classe do fragment, que encapsula toda a lógica de download da imagem. Lembre-se de que você pode abrir o projeto de exemplo deste capítulo no Android Studio para facilitar seus estudos.

DownloadImagemLoaderFragment.java

```

public class DownloadImagemLoaderFragment extends Fragment {
    private ProgressDialog progress;
    private ImageView imgView;
    @Override

```

```

public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
    super.onCreateOptionsMenu(menu, inflater);
    inflater.inflate(R.menu.menu_refresh, menu);
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if(item.getItemId() == R.id.action_refresh) {
        progress();
        Log.d("livroandroid", "restartLoader()");
        getLoaderManager().restartLoader(0, null, new LoaderCallbacks());
    }
    return super.onOptionsItemSelected(item);
}

@Override
public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container,
    @Nullable Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.fragment_download_imagem, container, false);
    imgView = (ImageView) view.findViewById(R.id.img);
    Log.d("livroandroid", "frag onCreateView");
    setHasOptionsMenu(true);
    return view;
}

@Override
public void onActivityCreated(@Nullable Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);
    Log.d("livroandroid", "frag onActivityCreated");
    progress();
    // Inicializa o loader ou reconecta a um existente
    Log.d("livroandroid", "initLoader()");
    getLoaderManager().initLoader(0, null, new LoaderCallbacks());
}

private void progress() {
    imgView.setImageBitmap(null);
    progress = ProgressDialog.show(getActivity(), "Aguarde", "Fazendo o download...");
    progress.setCancelable(true);
    progress.setOnCancelListener(new DialogInterface.OnCancelListener() {
        @Override
        public void onCancel(DialogInterface dialog) {
            getLoaderManager().getLoader(0).stopLoading();
        }
    });
}
}

```



```

private void setBitmap(Bitmap imagem) {
    Log.d("livroandroid", "setBitmap");
    // Esconde o progress
    closeProgress();
    // Atualiza a imagem
    imageView.setImageBitmap(imagem);
}

private class LoaderCallbacks implements LoaderManager.LoaderCallbacks<Bitmap> {
    @Override
    public Loader<Bitmap> onCreateLoader(int id, Bundle args) {
        Log.d("livroandroid", "onCreateLoader");
        return new ImageLoader(getContext());
    }
    @Override
    public void onLoadFinished(Loader<Bitmap> loader, Bitmap data) {
        Log.d("livroandroid", "onLoadFinished");
        setBitmap(data);
    }
    @Override
    public void onLoaderReset(Loader<Bitmap> loader) {
        Log.d("livroandroid", "onLoaderReset");
    }
}

@Override
public void onDetach() {
    super.onDetach();
    Log.d("livroandroid", "frag onDetach");
    closeProgress();
}

public Context getContext() {
    return getActivity();
}

private void closeProgress() {
    if(progress != null && progress.isShowing()) {
        progress.dismiss();
        progress = null;
    }
}
}

```

Para este exemplo, vamos adicionar um botão de atualizar na action bar.



/res/menu/menu_refresh.xml

```

<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:tools="http://schemas.android.com/tools" tools:context=".MainActivity" >
    <item android:id="@+id/action_refresh"
          android:title="Refresh" android:icon="@drawable/ic_action_refresh"
          app:showAsAction="always" />
</menu>

```

Ao executar esse código, o download da imagem será feito e o resultado será o mesmo de antes. Novamente recomendo que você execute o exemplo e verifique as mensagens no LogCat, que devem ser como mostradas a seguir:

```

D/livroandroid? frag onCreateView() // Cria a view do fragment
D/livroandroid? frag onActivityCreated() // Neste método do ciclo de vida, o fragment
                                         // sabe que o método onCreate() da activity
                                         // terminou. Aqui é um bom lugar para iniciar
                                         // o loader
D/livroandroid? initLoader() // O loader será criado ou reconectado se
                             // já existir
D/livroandroid? onCreateLoader() // Se o loader não foi criado ainda, este
                                 // método deve retornar o loader
D/livroandroid? loader onStartLoading() // Dentro da classe do loader, este método é
                                         // chamado para iniciar a tarefa
D/livroandroid? loader loadInBackground() // Executa em background para buscar os dados
                                         // e deve retornar o resultado
D/livroandroid? onLoadFinished() // Recebe o retorno do loader na UI Thread
D/livroandroid? setBitmap() // Atualiza o ImageView

```

Esses logs mostram a execução perfeita do loader. Agora faça o seguinte teste. Depois de carregar a imagem na primeira vez, gire a tela para provocar uma mudança nas configurações do sistema, assim sabemos que a activity e o fragment serão destruídos e recriados. Ao girar a tela, as mensagens do LogCat devem ser como mostradas a seguir:

```

D/livroandroid: frag onDetach() // O fragment será desassociado da activity
D/livroandroid: frag onCreateView() // Depois de girar a tela, cria a view do
                                     // fragment
D/livroandroid: frag onActivityCreated() // O fragment sabe que o método onCreate() da
                                         // activity terminou
                                         // Reconecta ao loader, pois ele já existe
D/livroandroid: initLoader() // Recebe o retorno do loader na UI Thread
D/livroandroid: onLoadFinished() // Atualiza o ImageView
D/livroandroid: setBitmap()

```

Veja que o interessante deste exemplo é que, depois de girar a tela, ao chamar o método `initLoader()`, o `LoaderManager` sabe que o loader já retornou os dados e simplesmente entrega o resultado no método `onLoadFinished()`. O interessante é que não foi necessário salvar o estado manualmente nem reter um fragment.

Nota: o método `initLoader(id,args,callback)` vai iniciar o loader ou reconectar em um loader já existente. Caso esse loader já tenha executado, ele simplesmente vai entregar o resultado para a UI Thread chamando o método `onLoadFinished()`. Caso você queira executar o loader novamente, deve chamar o método `restartLoader(id,args,callback)`.

Outro teste que você deve fazer é girar a tela durante o download. Neste caso, as seguintes mensagens são impressas no LogCat. Vou comentar apenas as partes ainda não explicadas.

```
D/livroandroid? frag onCreateView()
D/livroandroid? frag onActivityCreated()
D/livroandroid? initLoader()
D/livroandroid? onCreateLoader()
D/livroandroid? loader onStartLoading()
D/livroandroid? loader loadInBackground()
// Aqui eu girei a tela, durante o download
D/livroandroid: frag onDetach()           // A activity e o fragment serão destruídos
D/livroandroid: frag onCreateView()       // Cria a view do fragment novamente
D/livroandroid: frag onActivityCreated()
D/livroandroid: initLoader()              // Reconecta ao loader, pois ele já existe
// Aguarda a execução do loader
D/livroandroid: onLoadFinished()          // Recebe o retorno do loader na UI Thread
D/livroandroid: setBitmap()
```

Note que o interessante de utilizar loaders é que o `LoaderManager` faz o gerenciamento do estado do loader, e inclusive pode reconectar no loader depois de uma troca de configurações do sistema.

Podemos ver que o método `getLoaderManager().initLoader(id,args,callback)` vai criar o loader chamando o método `onCreateLoader(id,args)` na interface de callback, mas, se o loader com esse identificador já existir, ele será reutilizado. Por isso, caso você queira executar novamente a tarefa, é preciso explicitamente no código chamar o método `getLoaderManager().restartLoader(id,args,callback)`. Neste exemplo que criamos adicionei um botão de atualizar na action bar conforme a figura 31.4, que mostra o resultado:



Figura 31.4 – Ação de atualizar na action bar.

Nota: o método `initLoader(id,args,callback)` vai iniciar o loader ou reconectar a um loader já existente. Se o loader já tiver executado, ele simplesmente vai entregar o resultado para a UI Thread chamando o método `onLoadFinished()`. Caso você queira executar a tarefa do loader novamente, é necessário chamar o método `restartLoader(id,args,callback)`.

Ainda sobre a classe de loader que criamos anteriormente, vamos sobrescrever mais alguns métodos referentes ao ciclo de vida do loader. Altere o código conforme demonstrado a seguir e preste atenção nos comentários.



ImagemLoader.java

```
public class ImagemLoader extends AsyncTaskLoader<Bitmap> {
    private static final String URL = "http://livroandroid.com.br/imgs/livro_android.png";
    public ImagemLoader(Context context) {
        super(context);
    }
    @Override
    protected void onStartLoading() {
        super.onStartLoading();
        Log.d("livroandroid", "loader onStartLoading()");
        // Chamado para iniciar o loader ou reconectar a um já existente
        // Precisa chamar o forceLoad() para executar o loader "loadInBackground"
        forceLoad();
    }
}
```

```

@Override
protected void onForceLoad() {
    super.onForceLoad();
    // Chamado ao executar o método forceLoad()
    Log.d("livroandroid", "loader onForceLoad()");
}

@Override
public Bitmap loadInBackground() {
    Log.d("livroandroid", "loader loadInBackground()");
    // Faz o download da imagem
    Bitmap bitmap = null;
    try {
        bitmap = Download.downloadBitmap(URL);
    } catch (Exception e) {
        Log.e("livroandroid", e.getMessage(), e);
    }
    return bitmap;
}

@Override
public void deliverResult(Bitmap data) {
    // Executa depois do loadInBackground(). Chamado antes de enviar o resultado
    // para a UI Thread
    Log.d("livroandroid", "loader deliverResult(), isStarted(): " + isStarted());
    if(isStarted() && !isReset()) {
        super.deliverResult(data);
    }
}

@Override
protected void onStopLoading() {
    // Chamado quando o loader será parado
    // Acontece quando o método onStop() é chamado na activity ou fragment,
    // ou se você executou o stopLoading()
    super.onStopLoading();
    Log.d("livroandroid", "loader onStopLoading()");
}

@Override
protected void onReset() {
    super.onReset();
    // Chamado caso o loader tenha sido cancelado pelo método reset()
    Log.d("livroandroid", "loader onReset()");
}

@Override

```

```

public void onCancelCeled(Bitmap data) {
    super.onCanceled(data);
    Log.d("livroandroid", "loader onCancelCeled()");
}
@Override
public void onContentChanged() {
    super.onContentChanged();
    // Chamado se a fonte de dados utilizada pelo loader foi alterada
    // neste caso a interface deve ser alterada para refletir os novos dados
    // Não é utilizado muito com web services, estude o CursorLoader para entender isso
    Log.d("livroandroid", "loader onContentChanged()");
}
}

```

Nota: este assunto é avançado. O objetivo do livro não é lhe proporcionar apenas uma primeira leitura, então espero que você leia novamente o assunto sempre que necessário.

Para você entender o que significa cada método do ciclo de vida, vamos mais uma vez mostrar na prática. Faça o download da imagem e depois pressione a tecla home do Android. Isso vai deixar a activity e o fragment em background. Ao clicar na tecla home, a seguinte mensagem deve aparecer no LogCat:

```
D/livroandroid: loader onStopLoading()
```

Dessa maneira, podemos ver que o método `onStopLoading()` está associado com o ciclo de vida da activity/fragment, e isso é muito interessante.

Agora clique no ícone da aplicação na tela **Home** do Android para trazer a activity novamente para o topo da pilha. Neste caso o método `onResume()` será chamado na activity/fragment, e as seguintes mensagens devem aparecer no LogCat:

```

D/livroandroid? loader onStartLoading()
D/livroandroid? loader loadInBackground()
D/livroandroid? loader onForceLoad()
D/livroandroid? loader deliverResult, isStarted(): true
D/livroandroid? onLoadFinished()
D/livroandroid? setBitmap()

```

Então podemos verificar que quando a activity volta a executar o loader é reiniciado. Isso é feito automaticamente pelo `LoaderManager`. Porém aqui se você percebeu temos um problema, uma vez que o método `onStartLoading()` simplesmente está mandando executar o loader porque o método `forceLoad()` é chamado. E por

isso, conforme os logs, a tarefa do loader executou novamente e uma consulta foi realizada na internet para buscar a imagem.

```
@Override
protected void onStartLoading() {
    super.onStartLoading();
    Log.d("livroandroid", "loader onStartLoading()");
    forceLoad();
}
```

Acabamos de aprender um detalhe importante sobre o funcionamento dos loaders. Para solucionar esse problema, temos de manualmente salvar o estado do loader e decidir se ele deve recarregar ou não os dados, de forma parecida com o que já estudamos sobre salvar o estado de uma activity ou fragment. O código-fonte a seguir mostra como fazer isso. Veja que o objeto `Bitmap` é armazenado em um atributo da classe `ImageLoader`, e no método `onStartLoading()` é tomada a decisão sobre executar a tarefa ou não.

```
public class ImageLoader extends AsyncTaskLoader<Bitmap> {
    private static final String URL = "http://livroandroid.com.br/imgs/livro_android.png";
    private Bitmap bitmap;
    public ImageLoader(Context context) {
        super(context);
    }
    @Override
    protected void onStartLoading() {
        super.onStartLoading();
        if(bitmap == null) {
            Log.d("livroandroid", "loader onStartLoading() >> forceLoad()");
            // Executa o loader
            forceLoad();
        } else {
            Log.d("livroandroid", "loader onStartLoading() >> deliverResult()");
            // Já contém os dados, apenas atualiza a interface
            deliverResult(bitmap);
        }
    }
}
```

Isso vai resolver o problema de recarregar o loader quando a activity que está em segundo plano voltar a ocupar o topo da pilha. Portanto, faça o teste novamente: depois do download, clique no botão **Home** para sair da activity. Depois clique no ícone da aplicação para abrir a activity novamente. Desta vez podemos ver as seguintes mensagens no LogCat:

```
D/livroandroid: loader onStartLoading() >> deliverResult()
D/livroandroid: loader deliverResult, isStarted(): true
D/livroandroid: loader super.deliverResult()
```

Conforme essas mensagens de log, podemos ver que o loader não executou novamente, pois o `Bitmap` está em memória, portanto ele apenas entregou o resultado chamando o método `deliverResult()`. Outro detalhe importante do loader é como cancelá-lo. Se você verificar o código do fragment que fizemos, foi adicionado o listener para cancelar o `ProgressDialog`, e neste caso chamamos o método `stopLoading()` do loader.

```
private void progress() {
    imageView.setImageBitmap(null);
    progress = ProgressDialog.show(getActivity(), "Aguarde", "Fazendo o download...");
    progress.setCancelable(true);
    progress.setOnCancelListener(new DialogInterface.OnCancelListener() {
        @Override
        public void onCancel(DialogInterface dialog) {
            getLoaderManager().getLoader(0).stopLoading();
        }
    });
}
```

Para testar o cancelamento da execução do loader, faça o download e enquanto o `ProgressDialog` estiver aberto clique no botão **Voltar**. Feito isso, podemos ver que o método `stopLoading()` é chamado, o que coloca o loader em modo de parado/stop. Vale ressaltar que existe o método `isStarted()` do loader, que pode controlar justamente este estado para verificar se o loader está iniciado ou se está parado. O problema é que, mesmo depois de parar o loader, o método `deliverResult()` será chamado quando o método `loadInBackground()` terminar a tarefa. Isso significa que, mesmo cancelando a tarefa, o resultado ainda é entregue para a UI Thread atualizar a view. Então novamente temos de controlar isso manualmente, portanto altere este código na classe `ImageLoader`:

```
@Override
public void deliverResult(Bitmap data) {
    // Executa depois do loadInBackground(). Chamado antes de enviar o resultado
    // para a UI Thread
    Log.d("livroandroid", "loader deliverResult, isStarted(): " + isStarted());
    if(isStarted() && !isReset()) {
        Log.d("livroandroid", "loader super.deliverResult(): " + bitmap);
        super.deliverResult(data);
    }
}
```


Com isso, antes de entregar o resultado do loader, é feita a verificação se ele está executando com o método `isStarted()` e também é verificado se ele não foi invalidado com o método `isReset()`. Depois de adicionar estas linhas de código na classe `ImageLoader`, teste o cancelar no `ProgressDialog`.

Outro detalhe importante é que um loader pode ser invalidado, ou como se diz no inglês fazer o reset. Isso é feito chamando o método `reset()`. Neste caso, o método `onReset()` do loader deve eliminar os objetos de memória, pois não serão mais utilizados.

```
@Override
protected void onReset() {
    super.onReset();
    // Chamado caso o loader tenha sido cancelado pelo método reset()
    Log.d("livroandroid", "loader onReset()");
    this.bitmap = null;
}
```

Então, fazendo um resumo, podemos dizer que os métodos `onStartLoading()` e `onStopLoading()` são chamados quando o loader é conectado e desconectado da `activity/fragment`, pois ele fica atrelado ao ciclo de vida da `activity`.

O método `onReset()` é chamado se o método `reset()` for chamado e significa que este loader deve eliminar os dados de memória. O conceito é parecido como o `onDestroy()` da `activity`. Para exemplificar, se você fizer o download da imagem e depois pressionar o botão voltar do Android, a `activity` será destruída, e neste caso as seguintes mensagens devem aparecer no `LogCat`:

```
D/livroandroid: loader onStopLoading()
D/livroandroid: onLoaderReset()
D/livroandroid: loader onReset()
D/livroandroid: frag onDetach()
```

Nesses logs, podemos ver que ao destruir a `activity` o loader também é invalidado, pois o método `onReset()` foi chamado. Nesse momento, a aplicação deve limpar os recursos para liberar a memória.

31.5 Opinião do autor

Vimos que a interface `Loader` contém duas classes que a implementam: `CursorLoader` e `AsyncTaskLoader`. Nos exemplos deste capítulo, estudamos a classe `AsyncTaskLoader`. Já a classe `CursorLoader` vamos estudar no capítulo 32, sobre `content provider`.

Eu particularmente acho loaders complicados demais. Vimos que é necessário conhecimentos avançados para dominá-los de verdade.

Novamente, para consultar web services ou executar qualquer tarefa em segundo plano, eu particularmente utilizo aquele método `startTask(task)`, conforme fizemos no aplicativo dos carros. É simples e funciona. É exatamente assim que faço no meu dia a dia.

Se for necessário manter o estado de uma thread que está executando durante a rotação da tela, creio que você consiga resolver isso com um fragment retido em memória, conforme explicado neste capítulo.

Mas, enfim, cabe a você decidir se utilizará loaders ou não em seus aplicativos.

31.6 Links úteis

Neste capítulo, estudamos alguns assuntos mais avançados sobre como executar tarefas em segundo plano e gerenciar o estado da aplicação.

Loaders foram criados para funcionar de forma integrada ao ciclo de vida da aplicação, porém você precisa entender o seu funcionamento para tirar total proveito e usufruir do potencial dessa API. Já no caso da `AsyncTask`, ela simplesmente fornece uma biblioteca simples de threads que facilita algumas tarefas, mas você deve controlar o estado da aplicação manualmente.

Ambas as APIs funcionam e são muito utilizadas, então cabe a você estudá-las e verificar qual vai atender as necessidades do seu projeto. Separei alguns links da documentação oficial para complementar seus estudos.

- **Processes and Threads**

<http://developer.android.com/guide/components/processes-and-threads.html>

- **Loaders**

<http://developer.android.com/guide/components/loaders.html>



CAPÍTULO 32

Agenda de contatos e content provider

Content providers, ou simplesmente provedores de conteúdo, fornecem uma interface padrão para que aplicações possam expor e consultar informações. Um exemplo de provedor de conteúdo é a API da agenda de contatos.

Neste capítulo, aprenderemos a consultar os contatos da agenda, assim como criar nosso próprio provedor de conteúdo no projeto dos carros.

32.1 Por que utilizar a classe `ContentProvider` “provedor de conteúdo”

O Android permite armazenar informações de diversas formas diferentes utilizando banco de dados, arquivos e o sistema de preferências. Contudo, geralmente essas informações ficam salvas dentro do pacote da aplicação e somente a aplicação que criou o banco de dados ou o arquivo pode ter acesso às informações. Ou seja, a informação é privada da aplicação e ninguém pode acessá-la.

Se for necessário expor essas informações para outras aplicações, é preciso criar um provedor de conteúdo (content provider), que é representado pela classe `android.content.ContentProvider`. Um provedor de conteúdo tem as seguintes características:

1. Permite que aplicações compartilhem informações com outras aplicações.
2. Apresenta uma interface padrão de consulta e inserção.
3. Encapsula o acesso ao SQLite ou a outra fonte de dados.

É muito importante entender que você só precisa criar um provedor de conteúdo caso tenha necessidade de expor informações para outras aplicações. Se você precisar de um banco de dados privado que seja acessado apenas pelo seu aplicativo, utilize um banco com SQLite, como já mostramos.

Talvez a melhor forma de entender o que é um provedor de conteúdo seja brincarmos um pouco com algum provedor de conteúdo nativo; por isso, vamos começar este capítulo construindo exemplos para lermos os contatos da agenda.

O Android tem uma série de provedores de conteúdos nativos, como por exemplo consultar os contatos da agenda, visualizar os arquivos, imagens e vídeos disponíveis no celular. Independentemente de se esse provedor é nativo do Android ou se é criado pelo desenvolvedor, a forma como a aplicação se comunica com um provedor de conteúdo é padronizada, ou seja, apresenta uma interface padrão.

Para ter uma ideia sobre com o que vamos brincar neste capítulo, a figura 32.1 mostra o projeto de exemplo **HelloContatos** executando no emulador. No aplicativo será possível inserir na agenda os contatos Mickey, Pateta e Donald, para depois listá-los na lista. Ao clicar em algum contato, ele será mostrado com o aplicativo padrão do Android.

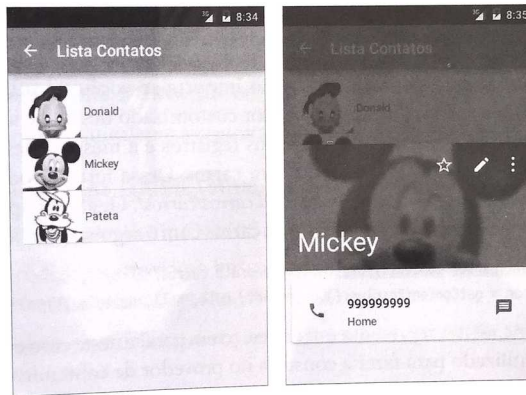


Figura 32.1 – Lista de contatos.

32.2 URI – Immutable URI reference

A classe `android.net.Uri` é utilizada em conjunto com um provedor de conteúdo para identificar um registro ou informação com uma string única e um formato amigável. Por exemplo, para identificar todos os contatos cadastrados na agenda do celular, podemos utilizar a URI `content://com.android.contacts/contacts/`. Para selecionar somente o contato com `id=1`, basta utilizar a URI `content://com.android.contacts/contacts/1`.

Com uma URI, é possível disparar uma consulta (query) para o Android, que por sua vez retornará um objeto do tipo `android.database.Cursor`, o qual permite ler o resultado da consulta. Já estudamos a classe `Cursor` no capítulo 18, sobre persistência e SQLite, portanto ela não é novidade. Este capítulo será moleza.

Veja este exemplo: o trecho de código demonstrado a seguir pode ser utilizado para buscar todos os contatos cadastrados na agenda. Utilizando o cursor retornado, é possível percorrer todos os contatos para ler informações.

```
Uri uri = Uri.parse("content://com.android.contacts/contacts/");
// query( Uri, projection, selection, selectionArgs, sortOrder)
Cursor cursor = getContentResolver().query(uri, null, null, null, null);
```

Se estamos interessados apenas no contato de `id=1`, podemos usar o seguinte trecho de código, que dessa vez retornará um cursor com apenas um registro:

```
Uri uri = Uri.parse("content://com.android.contacts/contacts/1"); // id=1
Cursor cursor = getContentResolver().query(uri, null, null, null, null);
```

Simples, não é? A ideia é muito interessante, e o mais impressionante é que tudo isso segue um padrão bem definido. Não importa se você está utilizando um provedor de conteúdo nativo, um provedor customizado desenvolvido por você ou de um terceiro, a maneira de buscar os registros é a mesma. Neste capítulo, criaremos um provedor de conteúdo para carros. Dessa forma, podemos dizer que a URI será `content://br.com.livroandroid.carros/carros/`. Qualquer aplicação que conheça essa URI poderá acessar a lista de carros com o seguinte trecho de código:

```
Uri uri = Uri.parse("content://br.com.livroandroid.carros/carro/");
Cursor cursor = getContentResolver().query(uri, null, null, null, null);
```

A classe `android.net.Uri` representa um endereço em geral e neste caso ela identifica o endereço utilizado para fazer a consulta no provedor de conteúdo.

32.3 Exemplos de provedores de conteúdo nativos

Conforme dito anteriormente, o Android tem uma série de provedores de conteúdo nativos que permitem consultar os contatos da agenda, visualizar os arquivos, imagens e vídeos disponíveis no celular, por exemplo.

A seguir, veja uma lista de algumas URIs que podem ser utilizadas para ler informações públicas do Android:

content://com.android.contacts/contacts/

URI que retorna todos os contatos cadastrados na agenda do celular.

content://com.android.contacts/contacts/1

URI que retorna apenas o contato com o id=1, se o mesmo estiver cadastrado na agenda do celular.

content://media/internal/images/media

URI que retorna todas as imagens disponíveis no celular.

content://media/external/images/media

URI que retorna todas as imagens disponíveis no cartão de memória.

Para que não seja necessário decorar cada string URI, é possível usar as classes utilitárias que fazem parte do provedor desejado. A lista a seguir exhibe as constantes que podem ser usadas no lugar de cada string descrita anteriormente:

`ContactsContract.Contacts.CONTENT_URI`

Constante equivalente a *content://com.android.contacts/contacts/*.

`MediaStore.Images.Media.INTERNAL_CONTENT_URI`

Constante equivalente a *content://media/internal/images/media*.

`MediaStore.Images.Media.EXTERNAL_CONTENT_URI`

Constante equivalente a *content://media/external/images/media*.

O seguinte código mostra como buscar todos os contatos da agenda utilizando a constante `ContactsContract.Contacts.CONTENT_URI`.

```
Cursor c = getContentResolver().query(ContactsContract.Contacts.CONTENT_URI, null,
    null, null, null);
```

Entretanto, para selecionar um único registro, como por exemplo selecionar o contato de id=1, é necessário utilizar o método `withAppendedId(uri, id)`, o qual recebe a URI padrão e o id do registro. O código a seguir mostra como selecionar o contato de id=1.

```
// Equivalente a "content://com.android.contacts/contacts/1"
Uri uri = ContentUris.withAppendedId(ContactsContract.Contacts.CONTENT_URI, 1);
Cursor c = getContentResolver().query(uri, null, null, null, null);
```

A classe `android.provider.ContactsContract` representa o provedor de conteúdo da agenda de contatos. Ela contém uma classe interna chamada `Contacts` que define a URI de consulta de contatos e as colunas disponíveis para consulta. Por exemplo, depois de consultar a agenda de contatos e obter o cursor, podemos ler o nome do contato desta forma:

```
Cursor c = . . . ;
String nome = c.getString(c.getColumnIndexOrThrow(ContactsContract.Contacts.DISPLAY_NAME));
```

32.4 Lendo os contatos da agenda

Para demonstrar o básico sobre a utilização de provedores de conteúdo, vamos criar um exemplo para ler os contatos da agenda. Crie um projeto no Android Studio chamado **HelloContatos** ou abra o projeto de exemplo do livro.

O projeto que vamos fazer vai ler e escrever na agenda de contatos, portanto declare as seguintes permissões no arquivo de manifesto:

```
<uses-permission android:name="android.permission.READ_CONTACTS" />
<uses-permission android:name="android.permission.WRITE_CONTACTS" />
```

O código a seguir mostra como fazer um rápido exemplo de como ler os contatos da agenda, portanto é só para aquecermos.



ListaContatosPrintActivity.java

```
public class ListaContatosPrintActivity extends AppCompatActivity {
    private static final String TAG = "livroandroid";
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_lista_contatos);
        printContatos();
    }
    private void printContatos() {
        // Uri: "content://com.android.contacts/contacts"
        Uri contatos = ContactsContract.Contacts.CONTENT_URI;
        Cursor cursor = getContentResolver().query(contatos, null,
            ContactsContract.Contacts.HAS_PHONE_NUMBER + " = 1 ", null, null);
        int count = cursor.getCount();
        Log.i(TAG, "Foram encontrados "+count+" contatos.");
        while(cursor.moveToNext()){
            String nome = cursor.getString(cursor.getColumnIndex(
                ContactsContract.Contacts.DISPLAY_NAME));
            Log.i(TAG, "Nome: " + nome);
        }
        // É importante fechar o cursor no final
        cursor.close();
    }
}
```

No código estou buscando todos os contatos que têm telefone, pois adicionei a condição `Contacts.HAS_PHONE_NUMBER + " = 1"`. Se você quiser, execute o código no seu celular, ou insira alguns contatos de exemplo no emulador para testar.

O resultado deste exemplo pode ser visto nos logs do LogCat. No caso do emulador eu adicionei três contatos, que foram impressos conforme o esperado.

```
INFO/ID(240): Foram encontrados três contatos
```

```
INFO/ID(240): Nome: Mickey
```

```
INFO/ID(240): Nome: Pateta
```

```
INFO/ID(240): Nome: Donald
```

Conforme visto neste exemplo, a classe `android.database.Cursor` permite que as informações retornadas sejam lidas da mesma forma que fizemos quando estudamos o SQLite. Para recuperar o cursor dos contatos, foi utilizado o método `getContentResolver().query(uri, projecao, selecao, selecaoArgs, orderBy)`, que recebe os seguintes argumentos:

Argumento	Descrição
Uri uri	URI para fazer a consulta; é o único parâmetro obrigatório.
String[] projecao	Array com os nomes das colunas para selecionar. Se for informado um array nulo, todas as colunas serão retornadas.
String selecao	String com uma cláusula SQL <code>WHERE</code> opcional para buscar alguma informação específica.
String[] selecaoArgs	Array com os argumentos utilizados pelo parâmetro <code>selecao</code> , se necessário.
String orderBy	String com um SQL para fazer o <code>order by</code> .

Veja que o método `getContentResolver()` retorna um objeto do tipo `ContentResolver`, que basicamente contém os métodos para acessar um provedor de conteúdo. Neste caso, utilizamos o método `query(...)`, mas existem outros como `insert(...)`, `update(...)` e `delete(...)`. Na prática, quando formos implementar nossa classe de provedor de conteúdo, vamos implementar todos esses métodos, pois o `ContentResolver` faz a ponte entre a aplicação e o provedor de conteúdo.

Dica: no projeto `HelloContatos` de exemplo deste capítulo, coloquei um botão na `action bar` para adicionar alguns contatos na agenda. Isso facilita os testes deste capítulo no emulador. Os contatos cadastrados são o Mickey, Pateta e Donald.

32.5 Como ler todos os telefones e a foto de um contato

Conforme demonstrado anteriormente, para ler um nome de um contato é utilizada a constante `Contacts.DISPLAY_NAME`.

O problema é que na classe `Contacts` não existe nenhuma constante para buscar os telefones ou a foto. Isso porque internamente essas informações são salvas em tabelas diferentes e consequentemente são outras classes que encapsulam esse acesso. Dessa forma, para ler todos os telefones de um contato, pode ser utilizado um método como este:

```
private List<String> getFones(Context context, long id) {
    List<String> fones = new ArrayList<String>();
    Cursor cursor = context.getContentResolver().query(
        ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null,
        ContactsContract.CommonDataKinds.Phone.CONTACT_ID + " = " + id,
        null, null);

    try {
        while (cursor.moveToNext()) {
            int coluna = cursor.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER);
            String fone = cursor.getString(coluna);
            fones.add(fone);
        }
    } finally {
        cursor.close();
    }
    return fones;
}
```

Note que a classe utilizada é a `ContactsContract.CommonDataKinds.Phone.CONTENT_URI`, que por sua vez contém a URI que internamente guarda a tabela do banco de dados com os telefones. Observe que como parâmetro da busca é informado o id do contato, com a constante `CONTACT_ID`.

Seguindo o mesmo raciocínio, podemos ler a foto de um contato, fazendo uma busca no provedor de conteúdo específico que guarda essa informação. Mas para ler uma foto existe um método utilitário dentro da classe `Contacts` que facilita a leitura. O trecho de código a seguir demonstra como ler a foto e obter um objeto `Bitmap` como retorno.

```
private Bitmap getFoto(Context context, long id) {
    // Cria a URI para o id fornecido
    Uri uri = ContentUris.withAppendedId(Contacts.CONTENT_URI, id);
    ContentResolver contentResolver = context.getContentResolver();
```

```

        InputStream in = ContactsContract.Contacts.
            openContactPhotoInputStream(contentResolver, uri);
        if (in != null) {
            Bitmap foto = BitmapFactory.decodeStream(in);
            return foto;
        }
        return null;
    }
}

```

Agora que já sabemos como ler o nome do contato, seus telefones e a sua foto, vamos juntar os conceitos nas classes Contato e Agenda:



Contato.java

```

public class Contato {
    public long id;
    public String nome;
    public Bitmap foto;
    public List<String> fones;
    // Retorna a URI deste contato, ex.: "content://com.android.contacts/contacts/1"
    public Uri getUri() {
        Uri uri = ContentUris.withAppendedId(Contacts.CONTENT_URI, id);
        return uri;
    }
    // Lê a foto de um contato
    public Bitmap getFoto(Context context) {
        // Cria a URI para o id fornecido
        Uri uri = ContentUris.withAppendedId(Contacts.CONTENT_URI, id);
        ContentResolver contentResolver = context.getContentResolver();
        InputStream in = Contacts.openContactPhotoInputStream(contentResolver, uri);
        if (in != null) {
            Bitmap foto = BitmapFactory.decodeStream(in);
            return foto;
        }
        return null;
    }
    public void show(Context context) {
        Uri uriContato = getUri();
        context.startActivity(new Intent(Intent.ACTION_VIEW, uriContato));
    }
}

```

Agenda.java

```
public class Agenda {
    // content://com.android.contacts/contacts
    private static final Uri URI = Contacts.CONTENT_URI;
    private static final String TAG = "agenda";
    private Context context;
    public Agenda(Context context) {
        this.context = context;
    }
    public List<Contato> getContatos() {
        // Recupera o cursor para percorrer a lista de contatos
        Cursor c = getCursorContatos();
        return getContatos(c);
    }
    public Cursor getCursorContatos() {
        return context.getContentResolver().query(URI, null, ContactsContract.Contacts.
            HAS_PHONE_NUMBER + " = 1 ", null, Contacts.DISPLAY_NAME);
    }
    public List<Contato> getContatos(Cursor cursor) {
        List<Contato> contatos = new ArrayList<Contato>();
        try {
            while (cursor.moveToNext()) {
                Contato a = getContato(cursor);
                contatos.add(a);
            }
        } finally {
            // Fecha o cursor
            cursor.close();
        }
        return contatos;
    }
    public Contato getContatoById(Long id) {
        Uri uri = ContentUris.withAppendedId(Contacts.CONTENT_URI, id);
        Cursor cursor = context.getContentResolver().query(uri, null, "has_phone_number=1",
            null, null);
        if (cursor.moveToNext()) {
            Contato c = getContato(cursor);
            return c;
        }
        return null;
    }
}
```

```

public Contato getContato(Cursor cursor) {
    Contato c = new Contato();
    // Id e nome
    long id = cursor.getLong(cursor.getColumnIndexOrThrow(Contacts._ID));
    c.id = id;
    String nome = cursor.getString(cursor.getColumnIndexOrThrow(Contacts.DISPLAY_NAME));
    c.nome = nome;
    // Fone
    boolean temFone = "1".equals(cursor.getString(cursor.getColumnIndexOrThrow(
        Contacts.HAS_PHONE_NUMBER)));
    if (temFone) {
        List<String> fones = getFones(id);
        c.fones = fones;
    }
    return c;
}

// Busca os telefones na tabela 'ContactsContract.CommonDataKinds.Phone'
private List<String> getFones(long id) {
    List<String> fones = new ArrayList<String>();
    Cursor cursor = context.getContentResolver().query(
        ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null,
        ContactsContract.CommonDataKinds.Phone.CONTACT_ID + " = " + id,
        null, null);

    try {
        while (cursor.moveToNext()) {
            int coluna = cursor.getColumnIndex(ContactsContract.CommonDataKinds.
                Phone.NUMBER);
            String fone = cursor.getString(coluna);
            fones.add(fone);
        }
    } finally {
        cursor.close();
    }
    return fones;
}

public boolean addContato(String nome, String telefone, int imgFoto) {
    try {
        // Lista de operações para executar no provedor de conteúdo
        ArrayList<ContentProviderOperation> operation =
            new ArrayList<ContentProviderOperation>();
        int backRefIndex = 0;
        // Adiciona o contato

```

```

operation.add(
    ContentProviderOperation.newInsert(ContactsContract.RawContacts.CONTENT_URI).
        withValue(ContactsContract.RawContacts.ACCOUNT_TYPE, null).
        withValue(ContactsContract.RawContacts.ACCOUNT_NAME, null).build());
// Nome do contato
operation.add(
    ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI).
        withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, backRefIndex).
        withValue(ContactsContract.Data.MIMETYPE,
            ContactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE).
        withValue(ContactsContract.CommonDataKinds.StructuredName.DISPLAY_NAME,
            nome).build());
// Associa o telefone do tipo "Home" ao contato
operation.add(
    ContentProviderOperation.newInsert(
        ContactsContract.Data.CONTENT_URI).
        withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, backRefIndex).
        withValue(ContactsContract.Data.MIMETYPE,
            ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE).
        withValue(ContactsContract.CommonDataKinds.Phone.NUMBER, telefone).
        withValue(ContactsContract.CommonDataKinds.Phone.TYPE,
            ContactsContract.CommonDataKinds.Phone.TYPE_HOME).build());
// Foto do contato
Bitmap fotoBitmap = BitmapFactory.decodeResource(context.getResources(), imgFoto);
ByteArrayOutputStream stream = new ByteArrayOutputStream();
fotoBitmap.compress(Bitmap.CompressFormat.PNG, 75, stream);
operation.add(
    ContentProviderOperation.newInsert(ContactsContract.Data.CONTENT_URI)
        .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID, backRefIndex)
        .withValue(ContactsContract.Data.IS_SUPER_PRIMARY, 1)
        .withValue(ContactsContract.Data.MIMETYPE,
            ContactsContract.CommonDataKinds.Photo.CONTENT_ITEM_TYPE)
        .withValue(ContactsContract.CommonDataKinds.Photo.PHOTO, stream.toByteArray())
        .build());
// Executa a lista de operações em batch
context.getContentResolver().applyBatch(
    ContactsContract.AUTHORITY, operation);
return true;
} catch (Exception e) {
    Log.d(TAG, "Erro ao inserir contato: " + e.getMessage(), e);
}
return false;
}

```

```

public int delete(Contato c) {
    return delete(c.id);
}

public int delete(Long id) {
    Uri uri = ContentUris.withAppendedId(Contacts.CONTENT_URI, id);
    int count = context.getContentResolver().delete(uri,null,null);
    return count;
}
}

```

Os métodos da classe Agenda estão comentados, portanto faça uma boa leitura no código. Utilizar essa classe é simples, o seguinte código mostra como ler os contatos:

```

Agenda a = new Agenda(this);    // this = context
List<Contato> contatos = a.getContatos();

```

Já para ler um contato pelo id, basta este código:

```

Agenda a = new Agenda(this);    // this = context
Contato c = a.getContatoById(id);

```

Como eu disse antes na dica, o projeto de exemplo HelloContatos está adicionando três contatos na agenda para testar. A seguir, podemos ver como a classe Agenda que criamos facilita esse trabalho. Mais uma vez, veja que gosto de criar classes utilitárias para tudo, pois acredito que seja mais simples utilizá-las em vez de chamar a API nativa.

```

public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.action_add) {
        Agenda a = new Agenda(this);
        a.addContato("Mickey", "999999999", R.drawable.mickey);
        a.addContato("Pateta", "888888888", R.drawable.pateta);
        a.addContato("Donald", "777777777", R.drawable.donald);
        Toast.makeText(this, "Contatos adicionados com sucesso.", Toast.LENGTH_SHORT).show();
        return true;
    }
    return super.onOptionsItemSelected(item);
}

```

32.6 Mostrando os contatos em um ListView

Para melhorar o primeiro exemplo que fizemos, o qual imprimiu os nomes dos contatos no LogCat, vamos criar um layout com um `ListView` a fim de mostrar a foto e nome do contato na lista. Ao clicar no contato, você poderá fazer algo com ele, mas no nosso caso vamos disparar uma intent para visualizar os detalhes do contato.

Vamos começar pelo layout da activity, o qual deve conter o ListView.



/res/layout/activity_lista_contatos.xml

```
<RelativeLayout . . .>
    <ListView android:id="@+id/listView"
        android:layout_width="match_parent" android:layout_height="match_parent" />
</RelativeLayout>
```

No código da activity, vamos ler todos os contatos e mostrá-los na lista.



ListaContatosPrintActivity.java

```
public class ListaContatosActivity extends AppCompatActivity
    implements AdapterView.OnItemClickListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_lista_contatos);
        final ListView listView = (ListView)
            findViewById(br.com.livroandroid.contatos.R.id.listView);
        listView.setOnItemClickListener(this);
        // Lista os contatos
        final Agenda a = new Agenda(this);
        final List<Contato> contatos = a.getContatos();
        final ContatoAdapter adapter = new ContatoAdapter(getBaseContext(), contatos);
        listView.setAdapter(adapter);
    }
    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
        Agenda a = new Agenda(this);
        Contato c = a.getContatoById(id);
        Toast.makeText(this, "Nome: " + c.nome, Toast.LENGTH_SHORT).show();
        c.show(this); // Mostra o contato disparando uma intent
    }
}
```

Para o código compilar, precisamos criar o adapter que vai mostrar o contato.



/res/layout/adapter_contato.xml

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="?android:attr/listPreferredItemHeight"
    android:gravity="center_vertical" android:orientation="horizontal">
```

```

<QuickContactBadge android:id="@+id/img"
    android:layout_width="72dp" android:layout_height="72dp" />
<TextView
    android:id="@+id/tNome" android:layout_marginLeft="4dp"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:gravity="center" />
</LinearLayout>

```

ContatoAdapter.java

```

public class ContatoAdapter extends BaseAdapter {
    private final LayoutInflater inflater;
    private Context context;
    private List<Contato> contatos;
    public ContatoAdapter(Context context, List<Contato> contatos) {
        this.context = context;
        this.contatos = contatos;
        this.inflater = LayoutInflater.from(context);
    }
    @Override
    public int getCount() {
        return contatos != null ? contatos.size() : 0;
    }
    @Override
    public Object getItem(int position) {
        return contatos.get(position);
    }
    @Override
    public long getItemId(int position) {
        Contato c = contatos.get(position);
        return c.id;
    }
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        View view = inflater.inflate(br.com.livroandroid.contatos.R.layout.
            adapter_contato,parent, false);
        TextView tNome = (TextView) view.findViewById(br.com.livroandroid.contatos.R.id.tNome);
        QuickContactBadge img = (QuickContactBadge) view.findViewById(br.com.livroandroid.
            contatos.R.id.img);
        Contato c = contatos.get(position);
        Uri uriContato = c.getUri();
        tNome.setText(c.nome); // Nome
        img.assignContactUri(uriContato); // Foto
    }
}

```



```
Picasso.with(context).load(uriContato).into(img);  
return view;  
}  
}
```

O interessante deste exemplo é que, em vez de utilizar um `ImageView` para mostrar a foto do contato, estamos utilizando a view `QuickContactBadge` no layout que facilita atribuir a foto do contato por meio de uma URI e ainda mostra o contato automaticamente ao clicar na foto. Dessa forma, não precisamos nos preocupar em como obter o `Bitmap` da foto do contato, pois o `QuickContactBadge` faz isso automaticamente, com uma ajudazinha da biblioteca Picasso, é claro.

Ao executar o código, o resultado deve ser como a figura 32.2. O lado direito da figura mostra os detalhes do contato logo depois de selecioná-lo.

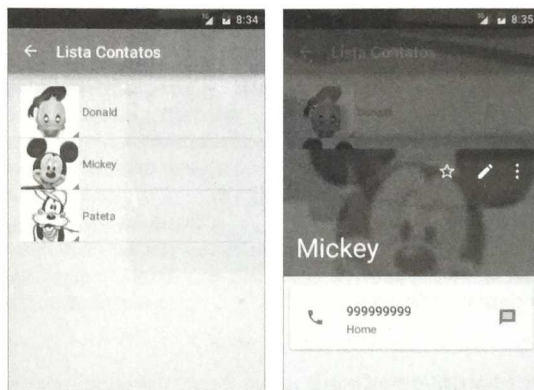


Figura 32.2 – Lista de contatos.

32.7 Utilizando um `CursorAdapter`

A implementação que fizemos tem um problema, pois ela carrega todos os contatos de uma vez só em memória; assim, caso existam muitos contatos na agenda, a aplicação pode demorar em exibir a lista.

Outro problema que temos é que eu li a agenda de contatos diretamente na UI Thread, sendo que o correto seria abrir uma thread ou async task para isso. Mas para simplificar o código deixei assim mesmo. Em outro tópico, vamos praticar a API de Loaders e resolver esse problema de forma simples.

Mas vamos voltar ao problema inicial deste tópico, que era carregar toda a lista em memória de uma só vez. Como isso é ruim, uma forma de resolver o problema seria fazer consultas paginadas que fossem buscando mais contatos à medida que o usuário fizesse o scroll na lista. Porém, isso seria um pouco complexo para demonstrar no livro, então vamos deixar para lá. Justamente para solucionar esse problema existe a classe `CursorAdapter`, que facilita criar um adapter plugado nos dados de um cursor, e uma de suas subclasses mais famosas é a `SimpleCursorAdapter`.

O código a seguir mostra como exibir os contatos em um `ListView` de forma simples utilizando o `SimpleCursorAdapter`. Nos parâmetros, note que precisamos passar o contexto, o layout do adapter, o cursor, os campos que devem ser lidos do cursor e os identificadores das views que estão no layout.

```
ListView listView = . . . ;
Cursor cursor = getContentResolver().query(. . . );
final SimpleCursorAdapter adapter = new SimpleCursorAdapter(
    getBaseContext(), // contexto
    R.layout.adapter_contato, // layout
    cursor, // cursor
    new String[]{ContactsContract.Contacts.DISPLAY_NAME}, // campos
    new int[]{R.id.tNome}, // campos no layout
    0);
listView.setAdapter(adapter);
```

Se você quiser testar esse código, fique à vontade. Porém, não gosto de fazer dessa forma, então vamos logo ao próximo exemplo. Um dos motivos pelos quais não gosto de fazer assim é porque temos de ficar mapeando tudo utilizando arrays. Outra questão é que nesse caso não tem como mostrar a foto do contato, pois, como já estudamos, ela fica em uma tabela diferente da do contato.

Uma solução simples é criar nossa própria subclasse de `CursorAdapter`, a qual vamos chamar de `ContatoCursorAdapter`.



ContatoCursorAdapter.java

```
public class ContatoCursorAdapter extends CursorAdapter {
    public ContatoCursorAdapter(Context context, Cursor c) {
        super(context, c, 0);
    }
    @Override
    public View newView(Context context, Cursor cursor, ViewGroup parent) {
        View view = LayoutInflater.from(context).inflate(R.layout.adapter_contato, parent, false);
        return view;
    }
}
```

```

@Override
public void bindView(View view, Context context, Cursor cursor) {
    TextView tNome = (TextView) view.findViewById(R.id.tNome);
    QuickContactBadge img = (QuickContactBadge) view.findViewById(R.id.img);
    int idxId = cursor.getColumnIndex(ContactsContract.Contacts._ID);
    Long id = cursor.getLong(idxId);
    Contato c = new Agenda(context).getContatoById(id);
    Uri uriContato = c.getUri();
    tNome.setText(c.nome);
    img.assignContactUri(uriContato);
    Picasso.with(context).load(uriContato).into(img);
}
}

```

Basicamente, esse adapter precisa implementar dois métodos: o método `newView(...)` e o método `bindView(...)`. O método `newView(...)` precisa inflar a view utilizada pelo adapter e o método `bindView(...)` vai preencher as views com cursor recebido no parâmetro, sendo que o cursor já estará posicionado no índice do contato corrente na lista. A vantagem de utilizar a classe `CursorAdapter` ou uma de suas subclasses é que ela faz uma utilização eficiente do cursor e otimiza a rolagem. Os resultados serão lidos rapidamente, assim como a rolagem da lista será fluida.

Para utilizar esse novo adapter, atualize o código da activity conforme demonstrado a seguir. Desta vez, estamos utilizando a classe `Agenda` apenas para obter o cursor de contatos, logo depois estamos criando o adapter com esse cursor.

ListaContatosPrintActivity.java

```

public class ListaContatosActivity extends AppCompatActivity implements AdapterView.
OnItemClickListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_lista_contatos);
        final ListView listView = (ListView)
            findViewById(br.com.livroandroid.contatos.R.id.listView);
        listView.setOnItemClickListener(this);
        // Lista os contatos
        Agenda a = new Agenda(this);
        Cursor cursor = a.getCursorContatos();
        final ContatoCursorAdapter adapter = new ContatoCursorAdapter(this, cursor);
        listView.setAdapter(adapter);
    }
    @Override

```

```

    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
        // Mesmo código de antes aqui
    }
}

```

Ao executar esse exemplo, novamente a lista de contatos será exibida, porém o resultado será bem mais eficiente do que o exemplo anterior, pois o `CursorAdapter` faz um melhor uso dos recursos e memória; portanto, no caso de provedores de conteúdo, é recomendado utilizá-lo.

32.8 Utilizando um `CursorLoader`

Junto com a API de Loaders que já estudamos no livro, foi criada a classe `CursorLoader`, que é uma subclasse de `AsyncTaskLoader`, que por sua vez é uma subclasse de `Loader`. Se precisar relembrar o conceito de loader, leia o capítulo anterior.

A classe `CursorLoader` é uma implementação de `Loader` que facilita a leitura dos dados de um cursor preenchendo automaticamente a view do adapter. A principal vantagem é que utilizando loaders tudo é feito em segundo plano, de forma que a interface da tela não fica travada. Logo, sobre todos os exemplos que fizemos anteriormente, podemos dizer que estavam errados, pois acessaram a agenda de contatos na `UI Thread`. Já a API de Loaders resolve isso de forma simples e eficiente.

O código a seguir mostra como mostrar a lista de contatos utilizando o `CursorLoader`. Veja que o `ListView` utiliza o mesmo adapter que fizemos no tópico anterior, porém foi informado um cursor nulo no construtor. Quando o método `getSupportLoaderManager().initLoader(...)` é chamado, o loader é iniciado. No método `onCreateLoader(...)` o loader é criado, e é aqui que retornamos o `CursorLoader`. Já o método `onLoadFinished(...)` é chamado quando o loader terminou de carregar os dados, ou seja, o cursor já foi lido. Neste caso, o método `adapter.swapCursor(cursor)` associa o adapter com o cursor para mostrar os dados na lista.

ListaContatosPrintActivity.java

```

public class ListaContatosActivity extends AppCompatActivity
    implements AdapterView.OnItemClickListener, LoaderManager.LoaderCallbacks<Cursor> {
    private static final String TAG = "livroandroid";
    private ListView listView;
    private CursorAdapter adapter;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_lista_contatos);
    }
}

```

```

        listView = (ListView) findViewById(R.id.listView);
        listView.setOnItemClickListener(this);
        // Configura o ListView com um adapter (cursor nulo)
        adapter = new ContatoCursorAdapter(this, null);
        listView.setAdapter(adapter);
        // Inicia o loader
        getSupportLoaderManager().initLoader(1, null, this);
    }

    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
        // Igual antes ...
    }

    @Override
    public Loader<Cursor> onCreateLoader(int id, Bundle args) {
        // Retorna o CursorLoader para carregar os contatos
        Uri uriContatos = ContactsContract.Contacts.CONTENT_URI;
        return new CursorLoader(getApplicationContext(), uriContatos,
            null, ContactsContract.Contacts.HAS_PHONE_NUMBER + " = 1 ", null,
            ContactsContract.Contacts.DISPLAY_NAME);
    }

    @Override
    public void onLoadFinished(Loader<Cursor> loader, Cursor cursor) {
        // Carrega o adapter com o cursor
        adapter.swapCursor(cursor);
    }

    @Override
    public void onLoaderReset(Loader<Cursor> loader) {
        // Limpa o adapter
        adapter.swapCursor(null);
    }
}

```

Ao executar esse código, novamente a lista com os contatos será exibida, mas desta vez utilizamos loaders, deixando a aplicação mais eficaz.

32.9 Monitorando a fonte de dados com um loader

Na explicação do capítulo 31, sobre `AsyncTask` e `Loader`, eu defendi a ideia de que loaders eram ligeiramente complexos, pelo menos na minha humilde opinião. Mas como o `CursorLoader` já está pronto, uma grande verdade é que ele realmente facilita nossa vida. Agora vamos entender o porquê. Com o próximo exemplo, você também entenderá o real significado de utilizar um loader.

Um dos maiores benefícios da API de Loaders é monitorar a fonte de dados para que, em caso de mudanças, a interface da tela possa ser atualizada rapidamente.

Para entender esse conceito na prática, siga os seguintes passos. Mostre os contatos na lista como fizemos anteriormente com qualquer um dos exemplos, menos o que utiliza loaders. Clique no contato para visualizar a tela de detalhes, depois edite o contato e exclua-o da agenda. Depois de excluir o contato, o Android vai voltar para a lista de contatos, porém o contato ainda estará lá! Você precisará fechar a tela e abri-la novamente para atualizar a lista, ou teríamos de implementar algum procedimento de atualização (refresh). Algo parecido com o que fizemos no projeto de carros, quando excluimos um carro do banco de dados e foi necessário atualizar a lista.

Agora refaça o mesmo teste com esta última implementação que utiliza loaders. Logo depois de excluir o contato e voltar à lista, automaticamente a lista estará atualizada, pois internamente o loader percebeu a mudança e informou ao adapter que era necessário atualizar a lista. Tudo acontece de forma transparente e simples.

Enfim, essa é uma das principais vantagens da API de Loaders. Para fechar este tópico e apenas revisando, seguem as principais características e benefícios de um loader:

1. Encapsula a AsyncTask, portanto executa com uma thread em segundo plano.
2. Facilita a atualização da view na UI Thread, pois a própria AsyncTask encapsula o uso de um handler.
3. Conforme já estudamos, o loader está atrelado ao ciclo de vida da activity e seus fragments.
4. O loader mantém o estado mesmo durante a troca de configurações de sistema, como a troca de orientação.
5. Conforme acabamos de aprender, um loader consegue monitorar a fonte de dados para que, em caso de mudanças, a interface da tela possa ser atualizada rapidamente.
6. Se no projeto dos carros tivéssemos um provedor de conteúdo com um loader, resolveríamos de forma simples o problema que tivemos ao excluir um carro no projeto. Na época tivemos que fazer um artifício técnico para atualizar a lista depois de excluir um carro do banco de dados. Se você tem curiosidade em aprender como um provedor de conteúdo é criado, continue lendo o próximo tópico.

32.10 Criando um provedor de conteúdo customizado

Agora que já sabemos o que é um provedor de conteúdo, vamos criar um no projeto dos carros. O objetivo é expor informações para outras aplicações, pois do contrário não vejo muita necessidade de criar um provedor de conteúdo devido a sua complexidade. A não ser que você queira resolver de forma elegante o problema 6 da lista do tópico anterior.

Então, mãos à obra! Para começar, vamos primeiro revisar a sintaxe de uma URI. Sabemos que para buscar os contatos da agenda são usadas as seguintes URIs:

- `content://com.android.contacts/contacts/`
- `content://com.android.contacts/contacts/1`

De forma similar, o provedor de conteúdo dos carros terá as seguintes URIs:

- `content://br.com.livroandroid.carros/carros`
- `content://br.com.livroandroid.carros/carros/1`

Observe que a sintaxe da URI segue o padrão `content://authority/path/id`, e na lista a seguir podemos verificar a explicação sobre cada parte:

Parte	Descrição
<i>prefix</i>	Início padrão, devendo ser sempre <code>content://</code> .
<i>authority</i>	Nome único que identifica um provedor de conteúdo. É recomendado que seja uma string com o pacote completo, para evitar duplicatas. A mesma string do <i>authority</i> é declarada no arquivo <i>AndroidManifest.xml</i> , no atributo <code>android:authorities</code> .
<i>path</i>	Caminho que o provedor de conteúdo usa para determinar o tipo do conteúdo que está sendo requisitado. No exemplo que será criado, o caminho para acessar o conteúdo é <code>/carros</code> . O caminho pode inclusive conter subcategorias, como <code>/carros/marca/modelo/</code> , desde que faça sentido para sua aplicação.
<i>record id</i>	Esta parte da URI é opcional e deve ser utilizada quando é necessário selecionar apenas um único registro. O id do registro, que corresponde à coluna “_id” do banco de dados, é fornecido nesse caso.

Algo importante sobre o provedor de conteúdo é a segunda parte, chamada de *authority*, pois quando formos declarar o provedor de conteúdo no *AndroidManifest.xml* precisamos informá-la, mas veremos isso no próximo tópico.

32.11 Classe ContentProvider

Chegou o grande momento: criaremos uma classe que será o provedor de conteúdo de carros, sendo assim vamos continuar o desenvolvimento do nosso projeto.

Crie um a classe CarroProvider conforme demonstrado a seguir.



CarroProvider.java

```
public class CarroProvider extends ContentProvider {
    @Override
    public String getType(Uri uri) {
        return null;
    }
    @Override
    public boolean onCreate() {
        return false;
    }
    @Override
    public Cursor query(Uri uri, String[] projection, String selection,String[]
        selectionArgs, String sortOrder) {
        return null;
    }
    @Override
    public Uri insert(Uri uri, ContentValues values) {
        return null;
    }
    @Override
    public int update(Uri uri, ContentValues values, String selection,String[]
        selectionArgs) {
        return 0;
    }
    @Override
    public int delete(Uri uri, String selection, String[] selectionArgs) {
        return 0;
    }
}
```

Antes de implementar os métodos, segue a explicação de cada um:

Método	Descrição
<code>String getType(Uri uri)</code>	Deve retornar o mime type para a URI informada. O valor retornado deve iniciar com <code>vnd.android.cursor.item</code> para um único registro, ou <code>vnd.android.cursor.dir/</code> para uma lista de registros. No exemplo que será construído para os carros, os tipos serão <code>vnd.android.cursor.item/vnd.google.carros</code> e <code>vnd.android.cursor.dir/vnd.google.carros</code> , respectivamente.
<code>boolean onCreate()</code>	Método chamado quando o provedor de conteúdo é criado. Deve retornar <code>true</code> se a inicialização foi realizada com sucesso, caso contrário, <code>false</code> .
<code>Cursor query(Uri uri, String[] projecao, String selecao, String[] selecaoArgs, String orderBy)</code>	Método usado para executar uma consulta, recebendo uma série de parâmetros. O método retorna um objeto <code>Cursor</code> que é utilizado para percorrer os resultados.
<code>Uri insert(Uri uri, ContentValues valores)</code>	Método utilizado para inserir um novo registro. O objeto <code>ContentValues</code> é um tipo de <code>HashMap</code> que armazena os nomes das colunas e seus respectivos valores para inserção.
<code>int update(Uri uri, ContentValues valores, String selecao, String[] selecaoArgs)</code>	Método utilizado para atualizar um registro. A URI fornecida deve ter o caminho completo para atualizar, incluindo o id do registro, por exemplo: <code>content://com.android.contacts/contacts/1</code> .
<code>int delete(Uri uri, String selecao, String[] selecaoArgs)</code>	Método utilizado para excluir um registro. A URI fornecida deve ter o caminho completo para excluir, incluindo o id, por exemplo: <code>content://com.android.contacts/contacts/1</code> .

Depois que a classe é criada, é necessário configurar a tag `<provider>` no arquivo *AndroidManifest.xml*.



AndroidManifest.xml

```
<!-- Content Provider -->
<provider
    android:name=".CarroContentProvider"
    android:authorities="@string/provider" android:exported="true" />
```

Para o código compilar, vamos criar o recurso `@string/provider`; mas, em vez de usar o arquivo *strings.xml* padrão, utilize o arquivo *strings_config.xml*, pois assim temos separadas as strings referentes às configurações.

```

/res/values/strings_config.xml

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="provider">br.com.livroandroid.carros</string>
    <!-- API KEY dos mapas -->
    <string name="API_KEY">AIzaSyDYCu8-Ijlg-dug-VzT15C_6fekDKn0oTQ</string>
</resources>

```

A lista a seguir explica o significado de cada item da tag <provider> que adicionamos no arquivo de manifesto.

- O atributo `android:name` recebe o nome da classe do tipo `ContentProvider`, da mesma forma que declaramos `activities`, `receivers` e `services`.
- O atributo `android:authorities` deve ser único e representa o pacote da URI do provedor de conteúdo.
- O atributo `android:exported` indica que o provedor de conteúdo pode ser acessado por outras aplicações, ou seja, você está expondo os dados do seu aplicativo. Na verdade, é esse o objetivo de um provedor de conteúdo.
- O atributo `android:enabled` é opcional e indica se o provedor de conteúdo está ativado ou não. O padrão obviamente é `true`.

Nota: podemos dizer que agora você já conhece o quarteto fantástico do Android. As classes `Activity`, `BroadcastReceiver`, `Service` e `ContentProvider` são todas cadastradas no arquivo de manifesto.

Agora vamos finalizar a implementação da classe `CarroProvider`. No código vamos deletar o trabalho de persistência para a classe `CarroDB` que já existe no projeto dos carros. Lembre-se de que não necessariamente o provedor de conteúdo precisa persistir os dados no SQLite, pois isso não importa.

```

CarroProvider.java

public class CarroContentProvider extends ContentProvider {
    // Classe que contém o banco de dados do carro
    private CarroDB db;
    // Colunas para selecionar
    private static HashMap<String, String> colunas;
    private static final int CARROS = 1;
    private static final int CARROS_ID = 2;
    // Utilizada para validar as expressões regulares sobre a URI

```

```

private UriMatcher uriCarro;
// Para criar a URI: content://br.com.livroandroid.carros/carros
private static String getAuthority() {
    return "br.com.livroandroid.carros";
}

public static final class Carros implements BaseColumns {
    // Não pode instanciar esta Classe
    private Carros() { }
    // content://br.com.livroandroid.carros/carros
    public static final Uri CONTENT_URI = Uri.parse("content://" + getAuthority() + "/carros");

    // Mime Type para todos os carros
    public static final String CONTENT_TYPE = "vnd.android.cursor.dir/vnd.google.carros";
    // Mime Type para um único carro
    public static final String CONTENT_ITEM_TYPE = "vnd.android.cursor.item/vnd.google.carros";
    // Ordenação default para inserir no order by
    public static final String DEFAULT_SORT_ORDER = "_id ASC";
    public static final String NOME = "nome";
    public static final String DESC = "desc";
    public static final String URL_INFO = "url_info";
    public static final String URL_FOTO = "url_foto";
    public static final String URL_VIDEO = "url_video";
    public static final String LATITUDE = "latitude";
    public static final String LONGITUDE = "longitude";
    public static final String TIPO = "tipo";
    // Método que constrói uma URI para um carro específico, com o seu id
    // A URI é no formato "content://br.livro.android.provider.carro/carros/id"
    public static Uri getUriId(long id) {
        // Adiciona o id na URI default do /carros
        Uri uriCarro = ContentUris.withAppendedId(Carros.CONTENT_URI, id);
        return uriCarro;
    }
}

@Override
public boolean onCreate() {
    // Uri Matcher para fazer as expressões regulares
    uriCarro = new UriMatcher(UriMatcher.NO_MATCH);
    // content://br.livro.android.provider.carro/carros
    uriCarro.addURI(getAuthority(), "carros", CARROS);
    // content://br.livro.android.provider.carro/carros/id
    uriCarro.addURI(getAuthority(), "carros/#", CARROS_ID);
    // Colunas para selecionar
    columns = new HashMap<String, String>();

```

```

        colunas.put(Carros._ID, Carros._ID);
        colunas.put(Carros.NOME, Carros.NOME);
        colunas.put(Carros.DESC, Carros.DESC);
        colunas.put(Carros.URL_INFO, Carros.URL_INFO);
        colunas.put(Carros.URL_FOTO, Carros.URL_FOTO);
        colunas.put(Carros.URL_VIDEO, Carros.URL_VIDEO);
        colunas.put(Carros.LATITUDE, Carros.LATITUDE);
        colunas.put(Carros.LONGITUDE, Carros.LONGITUDE);
        colunas.put(Carros.TIPO, Carros.TIPO);
        db = new CarroDB(getContext());
        return true;
    }

    @Override
    // Retorna o MIME type correto
    public String getType(Uri uri) {
        switch (uriCarro.match(uri)) {
            case CARROS:
                return Carros.CONTENT_TYPE;
            case CARROS_ID:
                return Carros.CONTENT_ITEM_TYPE;
            default:
                throw new IllegalArgumentException("URI desconhecida: " + uri);
        }
    }

    @Override
    public Uri insert(Uri uri, ContentValues initialValues) {
        // Valida se a URI é de /carros
        if (uriCarro.match(uri) != CARROS) {
            throw new IllegalArgumentException("URI desconhecida: " + uri);
        }
        ContentValues values;
        if (initialValues != null) {
            values = new ContentValues(initialValues);
        } else {
            values = new ContentValues();
        }
        long id = db.inset(values);
        if (id > 0) {
            // Inseriu
            Uri uriCarro = Carros.getUriId(id);
            getContext().getContentResolver().notifyChange(uriCarro, null);
            // Retorna a URI com o id do carro inserido
            return uriCarro;
        }
    }

```

```

    }
    throw new SQLException("Falhou ao inserir " + uri);
}
@Override
public Cursor query(Uri uri, String[] projection, String selection, String[]
    selectionArgs, String sortOrder) {
    // Classe utilitária para criar queries no SQLite
    SQLiteQueryBuilder builder = new SQLiteQueryBuilder();
    if(projection == null) {
        projection = new String[] {"_id", Carros.NOME, Carros.DESC, Carros.URL_INFO,
            Carros.URL_FOTO, Carros.URL_VIDEO, Carros.LATITUDE, Carros.LONGITUDE, Carros.TIPO};
    }
    // Configura a tabela para busca e a projeção
    switch (uriCarro.match(uri)) {
        case CARROS:
            builder.setTables("carro");
            builder.setProjectionMap(colunas);
            break;
        case CARROS_ID:
            builder.setTables("carro");
            builder.setProjectionMap(colunas);
            // where _id=?
            builder.appendWhere(Carros._ID + "=" + uri.getPathSegments().get(1));
            break;
        default:
            throw new IllegalArgumentException("URI desconhecida: " + uri);
    }
    // Ordenação
    String orderBy;
    if (TextUtils.isEmpty(sortOrder)) {
        orderBy = Carros.DEFAULT_SORT_ORDER;
    } else {
        orderBy = sortOrder;
    }
    // Query
    SQLiteDatabase db = this.db.getReadableDatabase();
    Cursor c = builder.query(db, projection, selection, selectionArgs, null, null, orderBy);
    // Notifica o content provider
    c.setNotificationUri(getContext().getContentResolver(), uri);
    return c;
}
@Override

```

```

public int update(Uri uri, ContentValues values, String where, String[] whereArgs) {
    int count;
    switch (uriCarro.match(uri)) {
        case CARROS:
            count = db.update(values, where, whereArgs);
            break;
        case CARROS_ID:
            // id do carro
            String id = uri.getPathSegments().get(1);
            // Atualiza o where se informado com o "_id=" para atualizar
            String whereFinal = Carros._ID + "=" + id + (!TextUtils.isEmpty(where)
                ? " AND (" + where + ')' : "");
            count = db.update(values, whereFinal, whereArgs);
            break;
        default:
            throw new IllegalArgumentException("URI desconhecida: " + uri);
    }
    // Notifica o content provider
    getContext().getContentResolver().notifyChange(uri, null);
    return count;
}

@Override
public int delete(Uri uri, String where, String[] whereArgs) {
    int count;
    switch (uriCarro.match(uri)) {
        case CARROS:
            count = db.delete(where, whereArgs);
            break;
        case CARROS_ID:
            // id do carro
            String id = uri.getPathSegments().get(1);
            // Atualiza o where se informado com o "_id=" para atualizar
            String whereFinal = Carros._ID + "=" + id + (!TextUtils.isEmpty(where)
                ? " AND (" + where + ')' : "");
            count = db.delete(whereFinal, whereArgs);
            break;
        default:
            throw new IllegalArgumentException("URI desconhecida: " + uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return count;
}
}
}

```

O código é extenso, mas não se assuste. Leia-o com calma e atentamente, pois basicamente foram implementados os métodos `query(...)`, `insert(...)`, `update(...)` e `delete(...)`, delegando a lógica para a classe `CarroDB`.

Os métodos da classe `ContentProvider` sempre recebem a URI, que em nosso exemplo será `content://br.com.livroandroid.carros/carros` ou `content://br.com.livroandroid.carros/carros/id` caso algum id seja fornecido para selecionar um carro em específico. Conforme a URI recebida, podemos descobrir se a informação que precisa ser retornada é uma lista de todos os carros ou apenas um carro específico. Para isso, a classe `android.content.UriMatcher` é utilizada. Observe que quase todos os métodos usam essa classe para verificar o tipo do registro a ser retornado.

Repare que em alguns locais do código é referenciada a parte `authority` da URI, pois apenas para relembrar a URI é composta das seguintes partes:

content://authority/path/id

A parte `authority` da URI deve ser cadastrada exatamente igual ao atributo `android:authorities` no arquivo de manifesto.

```
<!-- Content Provider -->
<provider
    android:name=".CarroContentProvider"
    android:authorities="br.com.livroandroid.carros" android:exported="true" />
```

32.12 Classe estática Carros

Algo importante sobre um provedor de conteúdo é que ele sempre tem uma classe interna que representa as colunas que as aplicações podem consultar ou inserir os dados.

No caso dos contatos, esta classe interna era a `ContactsContract.Contacts`, conforme podemos relembrar neste código:

```
Cursor c = . . . ;
String nome = c.getString(c.getColumnIndexOrThrow(ContactsContract.Contacts.DISPLAY_NAME));
```

Esta classe interna deve ser uma subclasse de `android.provider.BaseColumns`. Se você reparar no código da classe `CarroProvider` existe a classe interna `CarroProvider.Carros`.

```
public static final class Carros implements BaseColumns { . . .
    public static final String NOME = "nome";
    public static final String DESC = "desc";
    . . .
}
```

A classe `CarroProvider.Carros` possibilita acessar as colunas da tabela disponíveis para cada carro. Observe também que a classe `CarroProvider.Carros` define o método `getUriId(long id)` usado para construir uma URI informando o id do carro, por exemplo `content://br.com.livroandroid.carros/carros/fid`.

Concluindo, com as classes `CarroProvider` e `CarroProvider.Carros`, podemos consultar a lista de carros do content provider, conforme demonstrado a seguir:

```
// Busca os carros do nosso content provider: content://br.com.livroandroid.carros/carros
Cursor cursor = getContentResolver().query(CarroProvider.Carros.CONTENT_URI, null,
    null, null, null);
// Lê os carros do cursor
List<Carro> carros = new ArrayList<Carro>();
while (cursor.moveToNext()) {
    Carro c = new Carro();
    c.nome = cursor.getString(cursor.getColumnIndex(Carros.NOME));
    c.desc = cursor.getString(cursor.getColumnIndex(Carros.DESC));
    . . .
    carros.add(c);
}
// Fecha o cursor ao terminar de ler
cursor.close();
```

Como podemos ver, a forma de ler os dados no provedor de conteúdo segue o mesmo padrão, independentemente de qual seja.

Acredito que nossos estudos neste capítulo estão quase chegando ao fim. O provedor de conteúdos está criado no projeto dos carros, e agora uma forma de testar se tudo está funcionando é criar outro projeto separado e testar as consultas. Lembrando que o maior objetivo de um provedor de conteúdo é expor as informações para outra aplicação.

Não farei esse desenvolvimento aqui, pois este é um ótimo exercício para você. Apenas para auxiliá-lo, deixei pronto o projeto de exemplo **CarrosContentProvider**. Nesse projeto, criei a lista de carros consultando o provedor de conteúdo, da mesma forma que fizemos com o exemplo da agenda de contatos. Poderíamos até continuar o desenvolvimento para inserir, editar e excluir carros, mas isso fica como exercício para você.

32.13 Links úteis

Conforme estudamos neste capítulo, caso seja necessário expor as informações do nosso aplicativo para outras aplicações, podemos criar um provedor de conteúdo. O exemplo mais famoso de provedor de conteúdo é a agenda de contatos, pois essa aplicação permite que qualquer outra faça a leitura em sua base de dados.

Lembre-se de que, caso você não precise expor as informações do seu aplicativo, recomenda-se deixá-las privadas. Ou seja, é suficiente utilizar um banco de dados com SQLite, ou qualquer outro modelo de persistência.

Para complementar seus estudos, separei alguns links da documentação oficial.

- **API Guides – Content Providers**

<http://developer.android.com/guide/topics/providers/content-providers.html>

- **Android Training – Running a Query with a CursorLoader**

<https://developer.android.com/training/load-data-background/setup-loader.html>



CAPÍTULO 33

SMS

O Android permite enviar e receber uma mensagem SMS pelo aplicativo, e isso pode ser utilizado para fazer com que aplicações se comuniquem.

Embora o mundo mobile esteja cada vez mais conectado e dependente da conexão com a internet, às vezes enviar uma mensagem de SMS pode ser uma boa solução, pois não depende de conexão e existe a garantia de entrega da mensagem.

Neste capítulo, aprenderemos com exemplos práticos como enviar e receber mensagens por SMS.

33.1 Enviando SMS por intent ou pela API

Para aprendermos como enviar uma mensagem SMS, vamos continuar o exemplo do capítulo anterior que mostrava a lista de contatos, com nossos amigos: Mickey, Pateta e Donald.

Eu criei uma cópia do projeto anterior e chamei de **HelloContatosSMS**. Neste projeto deixei a MainActivity com o menu que insere os contatos Mickey, Donald e Pateta na agenda. A lista de contatos é mostrada pelo fragment ListaContatosFragment, que por sua vez utiliza o ListView com o CursorLoader, conforme aprendemos no capítulo anterior.

Abra esse projeto, pois ele será ponto de partida para os exercícios deste capítulo. Se quiser apenas verificar o resultado, o projeto **HelloContatosSMSOK** contém a solução final. Para começar, adicione as permissões SEND_SMS e RECEIVE_SMS no arquivo de manifesto, pois são necessárias para enviar e receber SMS.



AndroidManifest.xml

```
<manifest ... >
  <uses-permission android:name="android.permission.READ_CONTACTS" />
```

```

<uses-permission android:name="android.permission.WRITE_CONTACTS" />
<uses-permission android:name="android.permission.SEND_SMS" />
<uses-permission android:name="android.permission.RECEIVE_SMS" />
<application . . . />
</application>
</manifest>

```

Podemos enviar uma mensagem SMS por intent ou pela API. A classe `IntentUtils` da biblioteca `android-utils` contém um método para enviar o SMS por intent.

 `IntentUtils.java`

```

public class IntentUtils {
    public static void sendSms(Context context, String fone,String msg) {
        Uri uri = Uri.parse("sms:"+fone);
        Intent intent = new Intent(Intent.ACTION_SENDTO,uri);
        intent.putExtra("sms_body", msg);
        context.startActivity(intent);
    }
}

```

Para enviar o SMS pela API, crie a classe `Sms` com o seguinte código-fonte:

 `Sms.java`

```

public class Sms {
    private static final String TAG = "livro";
    // Envia um SMS para o número indicado
    public void send(Context context, String to, String msg) {
        try {
            SmsManager smsManager = SmsManager.getDefault();
            PendingIntent pIntent = PendingIntent.getBroadcast(context, 0, new Intent(), 0);
            smsManager.sendTextMessage(to, null, msg, pIntent, null);
            Log.d(TAG,"Sms.send to["+to+"] msg["+msg+"]");
        } catch (Exception e) {
            Log.e(TAG, "Erro ao enviar o SMS: " + e.getMessage(),e);
        }
    }
}

```

Uma vez que temos o código para enviar um SMS por intent ou pela API, vamos alterar o exemplo que mostra a lista de contatos, para que, ao selecionar um contato, seja exibido um popup com essas duas opções.

A lista de opções será criada como um array em XML.

 /res/values/strings.xml

```
<resources>
    . . .
    <string name="action_sms_intent">por Intent</string>
    <string name="action_sms_api">pela API</string>
</resources>
```

 /res/values/arrays.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <array name="popup_sms">
        <item>@string/action_sms_intent</item>
        <item>@string/action_sms_api</item>
    </array>
</resources>
```

Esse array pode ser inflado para `String[]`, sendo assim é melhor definir esses recursos em XML, pois facilitam a manutenção e a internacionalização.

Com o objetivo de inflar o array e mostrar um popup com as opções para o usuário escolher, vamos utilizar um `AlertDialog`. Siga em frente e altere o método `onItemClick()` que trata o evento de seleção da lista conforme demonstrado a seguir:

 ListaContatosFragment.java

```
public class ListaContatosFragment extends Fragment . . . {
    . . .
    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
        Agenda a = new Agenda(getActivity());
        final Contato c = a.getContatoById(id);
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
        builder.setTitle("SMS para: " + c.nome);
        builder.setItems(R.array.popup_sms, new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                String fone = c.fones.get(0);
                if(which == 0) { // envia SMS por intent
                    IntentUtils.sendSms(getActivity(), fone, "Olá "+c.nome+" , tudo bem?");
                } else { // envia SMS pela API
                    Sms sms = new Sms();
```

```

        sms.send(getActivity(),fone,"Olá "+c.nome+", tudo bem?");
        Toast.makeText(getActivity(), "SMS enviado para: " + c.nome,
            Toast.LENGTH_SHORT).show();
    }
}
});
AlertDialog dialog = builder.create();
dialog.show();
}
...
}

```

Feito isso, execute o projeto no emulador e clique em algum contato. A figura 33.1 mostra o alerta com as opções.

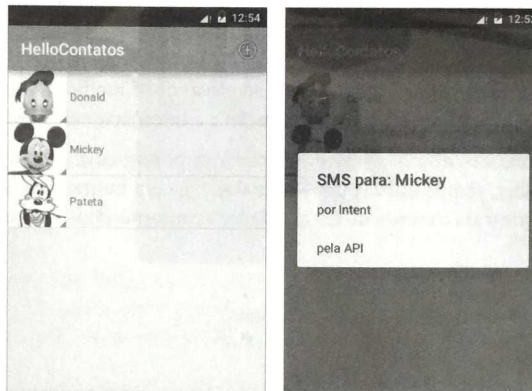


Figura 33.1 – Alerta ao selecionar o contato.

A figura 33.2 mostra o resultado do envio do SMS. O lado esquerdo da figura mostra a mensagem sendo enviada com uma intent. A vantagem de utilizar intents é que o aplicativo nativo é chamado, sendo assim o usuário pode redigir a mensagem e enviar o SMS com o aplicativo ao qual já está acostumado. O lado direito da figura mostra um toast logo após o SMS ter sido enviado pela API. A vantagem da API é que ela permite enviar mensagens sem interagir com o usuário.

Lembre-se de que é o usuário quem paga pelo envio das mensagens. Justamente por isso, é necessário declarar as permissões no *AndroidManifest.xml*, para avisá-lo de que o aplicativo pretende enviar mensagens.

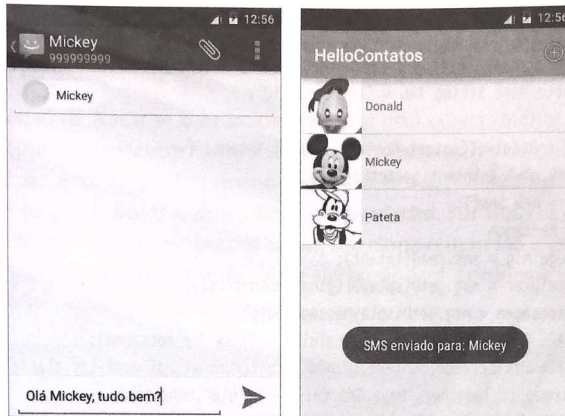


Figura 33.2 – Alerta ao selecionar o contato.

33.2 Criando um BroadcastReceiver para receber um SMS

Para receber um SMS no aplicativo, basta interceptar a mensagem de broadcast `SMS_RECEIVED`, a qual é enviada pelo sistema sempre que uma nova mensagem SMS é recebida pelo sistema.

Então, mãos à obra: declare a permissão `RECEIVE_SMS` no *AndroidManifest.xml* e configure um broadcast receiver para interceptar a ação `SMS_RECEIVED`.



AndroidManifest.xml

```
<manifest
. . .
  <uses-permission android:name="android.permission.RECEIVE_SMS" />
  <application . . . />
    <activity . . . >
      <receiver android:name=".SMSReceiver">
        <intent-filter>
          <action android:name="android.provider.Telephony.SMS_RECEIVED" />
          <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>
      </receiver>
    </application>
  </manifest>
```

A classe `SMSReceiver` pode ser visualizada a seguir:



SMSReceiver.java

```
public class SMSReceiver extends BroadcastReceiver {
    private static final String TAG = "livroandroid";
    @Override
    public void onReceive(Context context, Intent intent) {
        Log.d(TAG, ">" + intent.getAction());
        Sms sms = new Sms();
        // Lê a mensagem
        SmsMessage msg = sms.read(intent);
        String celular = msg.getDisplayOriginatingAddress();
        String mensagem = msg.getDisplayMessageBody();
        Log.d(TAG, "SMSReceiver: sms[" + celular + "] -> " + mensagem);
        NotificationUtil.create(context, 1, new Intent(context, MainActivity.class),
            R.mipmap.ic_launcher, "Novo SMS de: " + celular, mensagem);
    }
}
```

Esse receiver lê a mensagem SMS e a imprime no LogCat. Logo depois, mostra uma notificação ao usuário. Para ler a mensagem SMS, vamos criar o método `read(intent)` na classe `Sms` que criamos anteriormente.



Sms.java

```
public class Sms {
    ...
    // Lê uma mensagem da intent interceptada pela ação
    public SmsMessage read(Intent intent) {
        SmsMessage[] mensagens = getMessagesFromIntent(intent);
        if (mensagens != null) {
            return mensagens[0];
        }
        return null;
    }
    private SmsMessage[] getMessagesFromIntent(Intent intent) {
        Log.d(TAG, "Sms.getMessagesFromIntent: " + intent.getAction());
        Object pdus[] = (Object[]) (Object[]) intent.getSerializableExtra("pdus");
        SmsMessage msgs[] = new SmsMessage[pdus.length];
        for (int i = 0; i < pdus.length; i++) {
            msgs[i] = SmsMessage.createFromPdu((byte[])pdus[i]);
            String celular = msgs[0].getDisplayOriginatingAddress();
            String mensagem = msgs[0].getDisplayMessageBody();
        }
        return msgs;
    }
}
```

Este código lê a mensagem SMS que está no formato PDU (Protocol Description Unit) e retorna um array de objetos do tipo `SmsMessage`, o qual contém o remetente e o texto da mensagem recebida.

Com o objetivo de testar se o broadcast receiver está corretamente configurado para interceptar a mensagem SMS, podemos simular o envio de um SMS para o emulador. Para isso, abra a ferramenta **Tools > Android > Android Device Monitor**, selecione o emulador na janela **Devices** e abra a janela **EmulatorControl**. Por fim, digite o número do telefone fictício no campo **Incoming Number**, selecione o item **SMS** e escreva algum texto. Quando tudo estiver pronto, clique no botão **Send**, conforme podemos ver na figura 33.3.

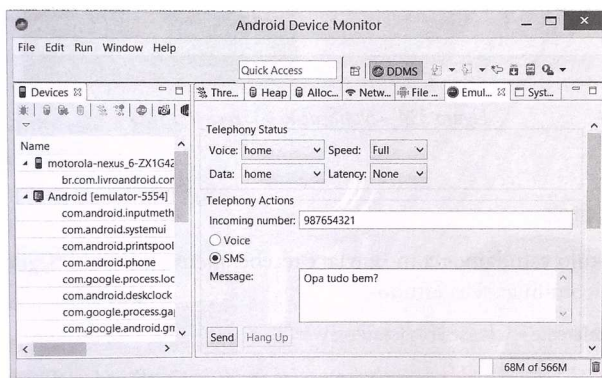


Figura 33.3 – Enviando um SMS para o emulador.

Isso vai simular o envio de um SMS. Sendo assim, caso o broadcast receiver esteja corretamente configurado, a aplicação vai interceptar a mensagem `SMS_RECEIVED` e vai imprimir no LogCat o remetente e o texto da mensagem.

```
D/livroandroid: SMSReceiver: sms[987654321] -> Opa, tudo bem?
```

A figura 33.4 mostra que existem dois ícones de notificação no emulador indicando que a mensagem foi recebida. O primeiro ícone é da aplicação nativa de mensagens e o segundo ícone é da nossa aplicação. Como a mensagem é enviada por broadcast, todos os receivers recebem a mensagem, portanto não tem como evitar que a aplicação nativa também a receba. Ao abrir a notificação (lado direito da figura), podemos ver as duas notificações, sendo que criei a notificação do livro seguindo o mesmo modelo da notificação nativa. O título é o número do remetente, e o conteúdo do texto é a mensagem recebida por SMS.

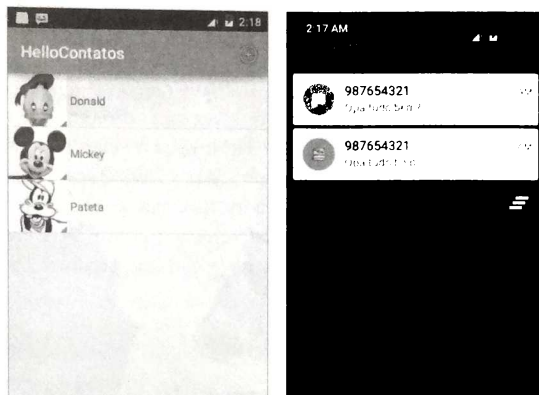


Figura 33.4 – Notificação ao receber o SMS.

33.3 Links úteis

Neste capítulo, estudamos como enviar e receber mensagens SMS. Seguem alguns links para continuar seus estudos.

- **API Reference** – Classe `BroadcastReceiver`
<http://developer.android.com/reference/android/content/BroadcastReceiver.html>
- **API Reference** – Classe `SmsManager`
<http://developer.android.com/reference/android/telephony/SmsManager.html>
- **API Reference** – Classe `SmsMessage`
<http://developer.android.com/reference/android/telephony/SmsMessage.html>



CAPÍTULO 34

Gestos

Gestos podem ser utilizados para reconhecer assinaturas digitais, desbloquear telas com um movimento cadastrado ou até reagir a simples gestos de swiipe lateral ou zoom.

Neste capítulo, vamos estudar os conceitos principais para implementar esse tipo de funcionalidade.

34.1 Introdução

O reconhecimento de gestos pode ser realizado de duas maneiras:

A primeira maneira é comparar o gesto efetuado pelo usuário com algum gesto previamente cadastrado. Esse é o caso de uma assinatura digital, na qual o usuário faz o cadastro da assinatura, a fim de repetir o gesto para validar a assinatura. A API permite comparar se o gesto feito pelo usuário segue o mesmo padrão da assinatura cadastrada e retorna uma pontuação (escore), informando se o gesto é parecido com o padrão salvo. Baseado na pontuação, a aplicação deve decidir se o gesto é válido ou não.

A segunda maneira de reconhecer gestos é monitorar o deslocamento nos eixos x e y, para reconhecer gestos bem populares, como o swiipe lateral e zoom.

34.2 Reconhecendo gestos previamente cadastrados

Antes de começarmos a brincar com gestos, vamos visualizar o exemplo que iremos criar, assim ficará mais fácil de entender o código.

Ao executar a aplicação pela primeira vez, você verá uma tela vazia, conforme a figura 34.1. Nesta tela você poderá desenhar um gesto, porém esse gesto só será reconhecido caso ele esteja cadastrado na biblioteca de gestos.

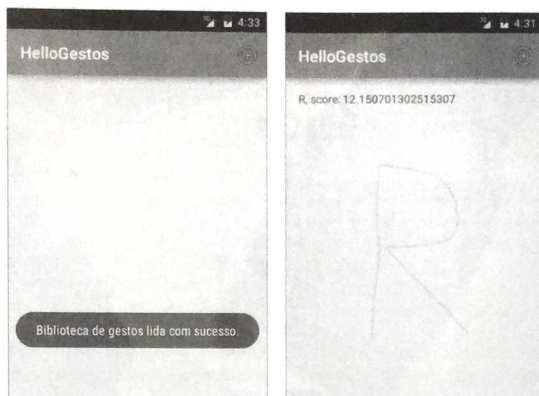


Figura 34.1 – Gesto reconhecido com sucesso.

Para cadastrar um gesto, clique no botão (+) na action bar. Isso vai abrir a tela para cadastrar um gesto, conforme a figura 34.2. Nessa tela, desenhe o gesto, digite um nome qualquer para ele e clique no botão **Salvar**. A figura 34.2 mostra um gesto com a letra R sendo salva na biblioteca.

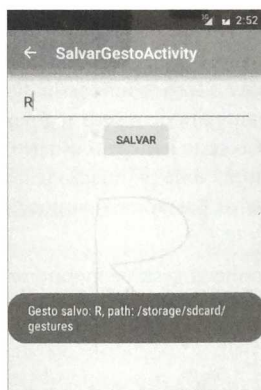


Figura 34.2 – Cadastrando um gesto.

Quando você já tiver cadastrado algum gesto em arquivo, volte à tela inicial e repita o mesmo gesto. O lado direito da figura 34.1 mostra o gesto da letra R sendo reconhecido com uma pontuação maior que 5.

A área que você pode desenhar o gesto é delimitada pela view `GestureOverlayView`, a qual foi inserida no centro do layout. Essa view sozinha faz o desenho e permite reconhecer o gesto. Depois de reconhecer um gesto, ele precisa ser validado. Isso é feito com a classe `GestureLibrary`, que permite carregar a biblioteca de gestos que foi salva em arquivo. Se o gesto for reconhecido como válido pela biblioteca, é mostrada a pontuação (escore) do gesto no `TextView`. Quanto mais alta for a pontuação, melhor, pois significa que o gesto desenhado é próximo do cadastrado. No nosso exemplo, estou aceitando somente gestos com pontuação superior a 5, pois é a aplicação que define qual pontuação mínima deseja aceitar.

Dica: na tela de cadastrar o gesto, você deve desenhar o gesto, digitar um nome para ele e clicar no botão **Salvar**. Foi o que fiz na figura 34.2. Somente gestos previamente cadastrados podem ser identificados. Depois de cadastrar o gesto, volte na tela inicial e tente repeti-lo. Se a pontuação (escore) for alta, o gesto será identificado.

Agora que já sabemos o que queremos fazer, vamos colocar a mão na massa. Crie um projeto chamado **HelloGestureLibrary** e adicione a permissão `WRITE_EXTERNAL_STORAGE` no arquivo de manifesto, pois vamos salvar os gestos feitos pelo usuário no SD card. Para reconhecer um gesto no aplicativo, deve-se utilizar a classe `android.gesture.GestureOverlayView`, que é uma view especial responsável por monitorar os gestos. Portanto, vamos criar o seguinte layout:



/res/layout/activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" android:padding="16dp">
    <TextView android:id="@+id/text"
        android:layout_width="match_parent" android:layout_height="wrap_content" />
    <FrameLayout android:id="@+id/layout" android:layout_weight="1"
        android:layout_width="match_parent" android:layout_height="0dp" >
        <ImageView android:id="@+id/img"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:layout_gravity="center"/>
        <android.gesture.GestureOverlayView android:id="@+id/gestureView"
            android:layout_width="match_parent" android:layout_height="match_parent"
            android:gestureColor="#00ff00" />
    </FrameLayout>
</LinearLayout>
```

Utilizei um `FrameLayout` para colocar o `GestureOverlayView` por cima do `ImageView`, assim logo depois de fazer o gesto é possível extrair o `Bitmap` do desenho com o objetivo de mostrá-lo no `ImageView`.

No código, para reconhecer os gestos, basta configurar o listener da classe `GestureOverlayView`:

```
GestureOverlayView gestureOverlayView = (GestureOverlayView) findViewById(R.id.gestureView);
gestureOverlayView.addOnGesturePerformedListener(this);
```

Quando um gesto for reconhecido, o método `onGesturePerformed(..)` será chamado, então podemos utilizar método `recognize(gesture)` da classe `GestureLibrary` para reconhecê-lo.

```
GestureLibrary gestureLib = . . . ;
@Override
public void onGesturePerformed(GestureOverlayView overlay, Gesture gesture) {
    // Faz a biblioteca de gestos reconhecer o movimento
    ArrayList<Prediction> predictions = gestureLib.recognize(gesture);
    . . .
}
```

A classe `GestureLibrary` representa a biblioteca de gestos conhecidos. Na prática a biblioteca de gestos é um arquivo que é salvo no SD card e pode até ser copiado para a pasta `/res/raw` do projeto. O trecho de código a seguir mostra como carregar a biblioteca de gestos a partir de um arquivo salvo na pasta `/res/raw`.

```
// Carrega a biblioteca de gestos a partir do arquivo /res/layout/gestures
GestureLibrary gestureLib = GestureLibraries.fromRawResource(this, R.raw.gestures);
boolean ok = gestureLib.load();
```

Isso é útil caso você queira embutir no aplicativo o arquivo previamente criado com a biblioteca de gestos. Mas geralmente os gestos serão salvos no SD card pelo aplicativo em tempo de execução. Neste caso podemos utilizar um código como este para ler a biblioteca e carregar o arquivo:

```
File file = new File(Environment.getExternalStorageDirectory(), "gestures");
gestureLib = GestureLibraries.fromFile(file);
boolean ok = gestureLib.load();
```

Será exatamente isso que vamos fazer no nosso exemplo. Para finalizar o exemplo, vamos mostrar o código da activity que carrega a biblioteca de gestos no método `onResume()` e faz o reconhecimento dos gestos.

 MainActivity.java

```

...
public class MainActivity extends AppCompatActivity implements OnGesturePerformedListener {
    private GestureLibrary gestureLib;
    private TextView text;
    private ImageView img;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        text = (TextView) findViewById(R.id.text);
        img = (ImageView) findViewById(R.id.img);
        // Configura o listener do GestureOverlayView
        GestureOverlayView gestureOverlayView = (GestureOverlayView)
            findViewById(R.id.gestureView);
        gestureOverlayView.addOnGesturePerformedListener(this);
    }
    @Override
    protected void onResume() {
        super.onResume();
        // Lê o arquivo de gestos do SD card
        readGestures();
    }
    @Override
    public void onGesturePerformed(GestureOverlayView overlay, Gesture gesture) {
        if(gestureLib == null) return;
        // Faz a biblioteca de gestos reconhecer o movimento
        ArrayList<Prediction> predictions = gestureLib.recognize(gesture);
        Prediction maxScore = null;
        for (Prediction prediction : predictions) {
            // Vamos aceitar somente escores maiores que 5
            if (prediction.score > 5.0) {
                if (maxScore == null || maxScore.score < prediction.score) {
                    maxScore = prediction;
                }
            }
        }
        // Se encontrou algum gesto com escore alto, vamos mostrar o texto
        if (maxScore != null) {
            // Se o escore é maior que 5
            String desc = maxScore.name + ", score: " + maxScore.score;

```

```

        text.setText(desc);
        // Mostra o gesto em um ImageView
        int w = (int) gesture.getBoundingBox().width();
        int h = (int) gesture.getBoundingBox().height();
        final Bitmap b = gesture.toBitmap(w, h, 8, Color.GREEN);
        img.setImageBitmap(b);
    }
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.menu_main, menu);
    return true;
}

@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.action_add) {
        startActivity(new Intent(this, SalvarGestoActivity.class));
        return true;
    }
    return super.onOptionsItemSelected(item);
}

public void readGestures() {
    File file = new File(Environment.getExternalStorageDirectory(), "gestures");
    if(file.exists()) {
        gestureLib = GestureLibraries.fromFile(file);
    }
    if (gestureLib != null && gestureLib.load()) {
        Toast.makeText(this, "Biblioteca de gestos lida com sucesso.",
            Toast.LENGTH_SHORT).show();
    }
}
}
}

```

Essa activity faz o reconhecimento dos gestos no método `onGesturePerformed(...)`. Logo depois que um gesto é detectado, é feito um loop por todos os objetos `Prediction` para verificar qual tem a maior pontuação. Se a pontuação (score) for acima de 5, o gesto é reconhecido, a pontuação é mostrada no `TextView` e o gesto é impresso no `ImageView`.

Para o código compilar, crie o arquivo de menu com o botão de adicionar (+).

 /res/menu/menu_main.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:tools="http://schemas.android.com/tools"
      xmlns:app="http://schemas.android.com/apk/res-auto" >
  <item android:id="@+id/action_add" android:title="@string/action_add"
        android:icon="@android:drawable/ic_menu_add" app:showAsAction="always" />
</menu>
```

Ao clicar no botão de adicionar (+), a activity SalvarGestoActivity será chamada; portanto, crie-a conforme o código-fonte demonstrado a seguir:

 /res/layout/activity_salvar_gesto.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:orientation="vertical" android:padding="16dp" >
  <EditText
    android:id="@+id/text"
    android:layout_width="match_parent" android:layout_height="wrap_content" />
  <Button
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal" android:text="@string/salvar"
    android:onClick="onClickSalvar"/>
  <FrameLayout android:id="@+id/layout"
    android:layout_width="match_parent" android:layout_height="0dp"
    android:layout_weight="1" >
    <ImageView android:id="@+id/img"
      android:layout_width="wrap_content" android:layout_height="wrap_content"
      android:layout_gravity="center"/>
    <android.gesture.GestureOverlayView android:id="@+id/gestureView"
      android:layout_width="match_parent" android:layout_height="match_parent"
      android:gestureColor="#00ff00" />
  </FrameLayout>
</LinearLayout>
```

Esse layout também reconhece o gesto com a classe `GestureOverlayView`, mas logo depois que o gesto é reconhecido ele permite digitar o nome do gesto e clicar no botão **Salvar** para salvá-lo na biblioteca de gestos em arquivo.



SalvarGestoActivity.java

```

...
public class SalvarGestoActivity extends AppCompatActivity implements GestureOverlayView.
OnGesturePerformedListener {
    private TextView text;
    private GestureLibrary gestureLib;
    private final File file = new File(Environment.getExternalStorageDirectory(), "gestures");
    private ImageView img;
    private Gesture gesture;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_salvar_gesto);
        text = (TextView) findViewById(R.id.text);
        img = (ImageView) findViewById(R.id.img);
        GestureOverlayView gestureOverlayView = (GestureOverlayView)
            findViewById(R.id.gestureView);
        gestureOverlayView.addOnGesturePerformedListener(this);
        // Carrega a biblioteca de gestos, a partir do arquivo salvo em /res/layout/gestures
        gestureLib = GestureLibraries.fromFile(file);
    }
    @Override
    public void onGesturePerformed(GestureOverlayView overlay, Gesture gesture) {
        // Mostra o gesto em um ImageView
        int w = (int) gesture.getBoundingBox().width();
        int h = (int) gesture.getBoundingBox().height();
        final Bitmap b = gesture.toBitmap(w, h, 8, Color.GREEN);
        img.setImageBitmap(b);
        overlay.setGesture(gesture);
        this.gesture = gesture;
    }
    public void onClickSalvar(View view) {
        String nome = text.getText().toString();
        if(gesture != null && nome != null) {
            // Salva o gesto
            gestureLib.addGesture(nome, gesture);
            gestureLib.save();
            final String path = file.getAbsolutePath();
            Toast.makeText(this, "Gesto salvo: " + nome + ", path: " + path,
                Toast.LENGTH_SHORT).show();
        }
    }
}

```

Capítulo 34 ■ Gestos

No caso do emulador, o arquivo com a biblioteca de gestos é salvo na pasta `/storage/sdcard/gestures`. Justamente por isso, no método `onResume()` da `MainActivity` esse arquivo é carregado em memória.

```
@Override
protected void onResume() {
    super.onResume();
    // Lê o arquivo da biblioteca de gestos
    readGestures();
}
```

Depois, no método `onGesturePerformed(..)`, os gestos detectados podem ser validados com base nesta biblioteca que foi carregada em memória.

```
ArrayList<Prediction> predictions = gestureLib.recognize(gesture);
```

Para testar o código, execute o projeto no emulador. O resultado deve ser como as figuras 34.1 e 34.2 explicadas no início deste tópico.

34.3 Detectando gestos comuns, como scroll lateral

Já aprendemos como utilizar a biblioteca de gestos para reconhecer gestos previamente cadastrados, mas às vezes o que precisamos fazer é apenas identificar um padrão de gesto amplamente conhecido, como um swiipe lateral ou zoom.

Para reconhecer esses gestos, podemos utilizar a classe `GestureDetector` e a interface `OnGestureListener`, a qual contém métodos para receber os eventos quando um gesto for detectado, conforme a lista a seguir:

```
abstract boolean onDown(MotionEvent e)
```

Chamado quando o evento de toque é iniciado.

```
abstract boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY)
```

Chamado quando um movimento de swiipe lateral foi encerrado, informando a coordenada inicial e a final, assim como a velocidade do movimento nos eixos x e y.

```
abstract void onLongPress(MotionEvent e)
```

Chamado se um toque demorado, por exemplo, com mais de um segundo, foi realizado.

```
abstract void onShowPress(MotionEvent e)
```

Chamado se o usuário fez o toque, mas ainda não soltou o dedo.

```
abstract boolean onSingleTapUp(MotionEvent e)
```

Chamado quando um toque simples é detectado.

No próximo exemplo, vamos demonstrar como utilizar o método `onFling(...)` para reconhecer o gesto de swipe lateral. Como precisamos implementar apenas um método, podemos criar uma subclasse de `SimpleOnGestureListener`, a qual já implementa todos esses métodos de forma vazia, assim podemos sobrescrever apenas o método pelo qual temos interesse.

Para continuar, crie o projeto `HelloGestos` e a seguinte activity. Se preferir abra o projeto de exemplo deste capítulo. Neste projeto criei duas activities: uma para reconhecer o gesto de swipe e outra para reconhecer o gesto de zoom.

Vamos começar pelo exemplo de gesto com rolagem lateral, conhecido como swipe.



SwipeActivity.java

```
...
import static com.nineoldandroids.view.ViewPropertyAnimator.animate;

public class SwipeActivity extends AppCompatActivity {
    TextView text;
    private ImageView img;
    private GestureDetector gestureDetector;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_swipe);
        img = (ImageView) findViewById(R.id.img);
        text = (TextView) findViewById(R.id.text);
        text.setText("Faça um gesto");
        // Cria a instância de GestureDetector e informa o listener
        gestureDetector = new GestureDetector(this, new MyFlingGestureDetector());
    }
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        // Aqui está o segredo. Delega o tratamento do touch para a classe GestureDetector
        boolean tratouEvento = gestureDetector.onTouchEvent(event);
        if (tratouEvento) {
            return tratouEvento;
        }
    }
}
```

```

        return super.onTouchEvent(event);
    }
    // Listener para reconhecer o gesto
    class MyFlingGestureDetector extends GestureDetector.SimpleOnGestureListener {
        // Distância do movimento em pixels. Por exemplo, o usuário tocou na tela e moveu o
        // dedo 100 pixels para a direita
        private float swipeMinDistance = 100;
        // Velocidade do movimento em pixels por segundo
        private float swipeThresholdVelocity = 200;
        @Override
        public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY) {
            // Usuário fez um gesto de swipe lateral
            try {
                if (e1.getX() - e2.getX() > swipeMinDistance && Math.abs(velocityX) >
                    swipeThresholdVelocity) {
                    // Fez um gesto da direita para a esquerda
                    text.setText("<<< Left");
                    animate(img).xBy(-100); // Move 100px para esquerda
                } else if (e2.getX() - e1.getX() > swipeMinDistance && Math.abs(velocityX) >
                    swipeThresholdVelocity) {
                    // Fez um gesto da esquerda para a direita
                    text.setText("Right >>>");
                    animate(img).xBy(+100); // Move 100px para direita
                }
            } catch (Exception e) { }
            return false;
        }
    }
}

```

 /res/layout/activity_swipe.xml

```

<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:background="#eeeeee" android:padding="16dp" >
    <ImageView
        android:id="@+id/img"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="top|center_horizontal"
        android:src="@drawable/android" />
    <TextView
        android:id="@+id/text" android:layout_gravity="center"

```

```

        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/faca_um_gesto" android:textSize="18sp" />
</FrameLayout>

```



/res/values/strings.xml

```

<resources>
    . . .
    <string name="faca_um_gesto">Faça um gesto</string>
</resources>

```

Esse arquivo de layout tem uma imagem no topo e um `TextView` no centro, o qual vamos utilizar para exibir as informações sobre o gesto. Por exemplo, se o usuário mover o dedo da esquerda para a direita, vamos mostrar essa informação no `TextView`.

Para o código compilar, adicione a biblioteca **NineOldAndroids** no arquivo *app/build.gradle*.



app/build.gradle

```

. . .
compile 'com.nineoldandroids:library:2.4.0'

```

Nota: ao fazer o swiipe lateral, estamos utilizando a biblioteca **NineOldAndroids** (<http://nineoldandroids.com/>) para movimentar a imagem de forma animada para esquerda ou direita, conforme o gesto efetuado. A biblioteca **NineOldAndroids** permite utilizar a sintaxe da `ViewPropertyAnimator` criada no Android 3.1 (Honeycomb) mantendo a compatibilidade com versões antigas do Android.

Ao executar o exemplo, você verá um texto escrito “faça um gesto” no centro da tela, conforme a figura 34.3. A figura ao lado mostra o resultado logo após fazer um gesto da esquerda para a direita. Conforme podemos visualizar, o resultado é que o texto **Right** foi exibido no centro da tela e a imagem foi deslocada para a direita, pois o gesto de swiipe lateral foi reconhecido com sucesso.

Depois de ver o exemplo em ação, vamos entender o código. O segredo é criar um objeto do tipo `GestureDetector` e uma implementação de `OnGestureListener`, para que o método `onFling(...)` seja chamado assim que o gesto for detectado.

Mas para a classe funcionar, você precisa invocá-la de alguma forma. Isso pode ser feito sobrescrevendo o evento de `onTouch(event)` da própria `activity` ou de alguma `view` customizada. Neste caso, estamos fazendo tudo direto na `activity`. Dessa forma, sempre que o usuário tocar na tela, a classe `GestureDetector` será invocada para detectar o gesto. Note que o método `onTouchEvent(event)` pode retornar `true`, indicando que o gesto foi detectado por essa classe, ou `false`, indicando o contrário.

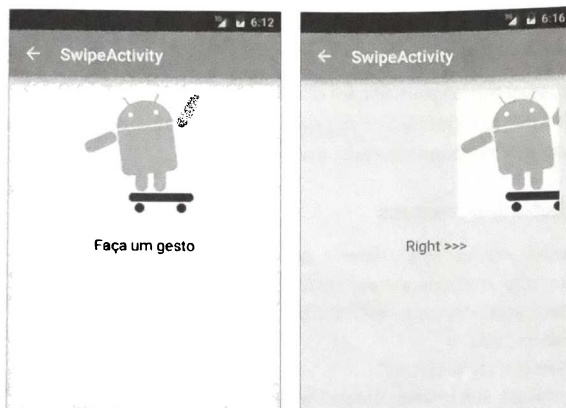


Figura 34.3 – Gesto de swipe.

Quando o gesto de movimento for detectado, o método `onFling(...)` é chamado. Observe que nos parâmetros recebemos dois objetos do tipo `MotionEvent`, que contêm as posições x/y de onde o gesto foi iniciado e finalizado, assim como a velocidade do gesto, medida em pixels por segundo.

Basicamente, o que precisamos fazer é verificar a distância percorrida no touch entre os pontos inicial e final do eixo x. Se a distância for maior que um valor configurável, significa que o usuário arrastou o dedo por uma distância suficiente para que o gesto se concretizasse.

Por fim, estamos movendo a imagem do Android para direita ou esquerda, conforme o gesto efetuado. Lembrando que para fazer a animação foi utilizada a biblioteca **NineOldAndroids**, por isso importamos a função `animate` de forma estática no código.

```
import static com.nineoldandroids.view.ViewPropertyAnimator.animate;
```

Com essa biblioteca, animar uma view (mantendo a compatibilidade) é simples assim:

```
animate(img).xBy(-100); // Move 100px para esquerda
```

34.4 Detectando gesto de pinch (zoom)

Outro gesto bastante comum é o de fazer zoom em alguma view, que consiste em tocar dois dedos na tela e fazer o gesto conhecido como pinch.

Esse gesto é utilizado em diversos aplicativos nativos do Android, como o álbum de fotos e o browser. Ao visualizar uma foto, podemos fazer o zoom da imagem

com o clássico gesto de pinch, assim como no browser, para dar zoom em alguma página que está sendo visualizada.

Como esse gesto é muito popular, foi criada a classe `ScaleGestureDetector`, que facilita muito o trabalho de detectá-lo. Para demonstrar, vamos criar a activity chamada `ZoomActivity` conforme demonstrado a seguir:



/res/layout/activity_zoom.xml

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="16dp" >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal" android:text="@string/faca_um_gesto" />
    <br.com.livroandroid.gestos.MyImageView
        android:id="@+id/img"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="center" />
</FrameLayout>
```

Note que a view `MyImageView` inserida no layout é customizada, portanto precisamos criar essa classe que será utilizada para desenhar uma imagem no centro da tela. Essa view será utilizada para fazer o desenho da imagem escalada logo após o gesto de zoom.



`MyImageView.java`

```
package br.com.livroandroid.gestos;
...
public class MyImageView extends View {
    private static final int img = R.drawable.android;
    private Drawable drawable;
    private float scaleFactor = 1;
    private int imgW;
    private int imgH;
    private Paint paint;
    public MyImageView(Context context) {
        super(context);
        init(context);
    }
}
```

```

    public MyImageView(Context context, AttributeSet attrs, int defStyle) {
        super(context, attrs, defStyle);
        init(context);
    }
    public MyImageView(Context context, AttributeSet attrs) {
        super(context, attrs);
        init(context);
    }
    private void init(Context context) {
        // Cria o drawable/imagem para desenhar
        drawable = context.getResources().getDrawable(ing);
        imgW = drawable.getIntrinsicWidth();
        imgH = drawable.getIntrinsicHeight();
        drawable.setBounds(0, 0, imgW, imgH);
        // Cria o pincel azul
        paint = new Paint();
        paint.setStyle(Style.STROKE);
        paint.setColor(Color.BLUE);
    }
    @Override
    public void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        // Desenha um quadrado apenas para visualizar a view
        canvas.drawRect(0, 0, getWidth() - 1, getHeight() - 1, paint);
        // Multiplica a largura e a altura pela escala
        int larguraComEscala = (int) (imgW * scaleFactor);
        int alturaComEscala = (int) (imgH * scaleFactor);
        // Posiciona o desenho no centro da tela
        int dx = getWidth() / 2 - larguraComEscala / 2;
        int dy = getHeight() / 2 - alturaComEscala / 2;
        canvas.translate(dx, dy);
        // Faz a escala do desenho (para dar zoom out/in)
        canvas.scale(scaleFactor, scaleFactor);
        // Desenha a imagem
        drawable.draw(canvas);
    }
    public void setScaleFactor(float scaleFactor) {
        // Permite à activity configurar a escala e redesenhar a view
        this.scaleFactor = scaleFactor;
    }
}

```


O código dessa classe está criando um `Drawable` com a imagem `/res/drawable/android.jpg` para depois no método `onDraw(canvas)` desenhar a imagem.

O código faz um cálculo para posicionar a imagem no centro da tela, algo que é bem comum em desenhos de baixo nível em `canvas`. Depois que descobrimos as posições `x` e `y` do centro da tela, o método `canvas.translate(dx,dy)` é chamado para informar ao `canvas` que o desenho deve ser feito nessas coordenadas.

Repare também que temos a chamada do método `canvas.scale(scaleX,scaleY)`, que faz o desenho ser escalado conforme a proporção informada para os eixos `x` e `y`. Por exemplo, se você escalar a imagem com o valor 1, significa que a imagem ficará com o tamanho original, já 0,5 corresponde à metade do tamanho, e 2, ao dobro do tamanho original.

Note que a escala da imagem é configurada pelo atributo `scaleFactor`, o qual por padrão tem o valor 1. Nossa brincadeira será fazer a `activity` alterar o valor dessa escala, conforme o gesto de zoom do usuário, refletindo no tamanho da imagem a fim de obter os efeitos de **zoom out** e **zoom in**.

Para finalizar o exemplo do zoom, vamos escrever o código da `activity`.



ZoomActivity.java

```
...
public class ZoomActivity extends AppCompatActivity {
    TextView text;
    private ScaleGestureDetector gestureDetector;
    private MyImageView img;
    private float scaleFactor = 1;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_zoom);
        // Informa o fator de escala padrão
        img = (MyImageView) findViewById(R.id.img);
        img.setScaleFactor(scaleFactor);
        text = (TextView) findViewById(R.id.text);
        text.setText("Faça um gesto de pinch/zoom");
        // Configura o detector de gestos
        gestureDetector = new ScaleGestureDetector(this, new ZoomGestureDetector());
    }
    @Override
    public boolean onTouchEvent(MotionEvent event) {
        // Delega o evento de touch ao detector de gestos
```

```

        boolean ok = gestureDetector.onTouchEvent(event);
        if(ok) {
            return super.onTouchEvent(event);
        }
        return ok;
    }
    // Detecta o zoom
    class ZoomGestureDetector extends ScaleGestureDetector.SimpleOnScaleGestureListener {
        @Override
        public boolean onScale(ScaleGestureDetector detector) {
            // Armazena o fator de escala detectado
            scaleFactor *= detector.getScaleFactor();
            // Exibe o fator de escala no TextView
            text.setText(String.valueOf(scaleFactor));
            // Altera o fator de escala da view customizada e pede para ela se redesenhar
            img.setScaleFactor(scaleFactor);
            img.invalidate();
            return true;
        }
        @Override
        public boolean onScaleBegin(ScaleGestureDetector detector) {
            return true;
        }
        @Override
        public void onScaleEnd(ScaleGestureDetector detector) {
        }
    }
}

```

Essa classe implementa a interface `SimpleOnScaleGestureListener` para receber o evento sempre que um gesto de pinch/zoom é feito pelo usuário. O método mais importante que precisamos implementar é o `onScale(detector)`, o qual é chamado quando o gesto de zoom é reconhecido. Nesse momento, a escala do gesto é recuperada pelo método `detector.getScaleFactor()` e armazenada na variável `scaleFactor`, a qual é passada para a view customizada. No método `onDraw(canvas)` da view esse fator de escala é utilizado para desenhar a view com o zoom definido pelo usuário.

O resultado disso é um exemplo interessante que vai exibir uma imagem no centro da tela. Você poderá fazer o gesto de **zoom out** e **zoom in** para brincar com a escala da imagem. A figura 34.4 mostra o resultado. A primeira parte mostra a imagem original com escala = 1 e ao lado podemos ver a imagem com o dobro do tamanho com uma escala próxima de 2; ou seja, fiz o gesto de zoom para expandir a imagem.

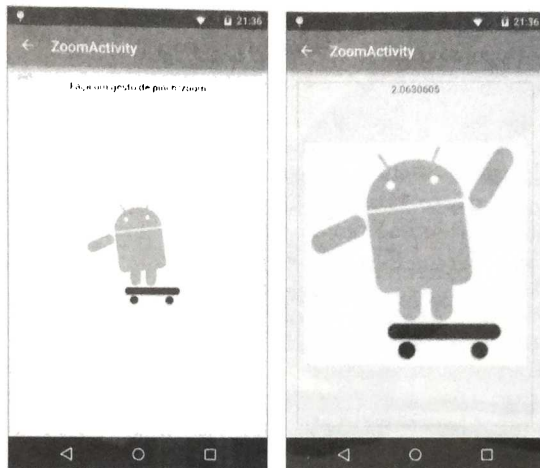


Figura 34.4 – Gesto de zoom.

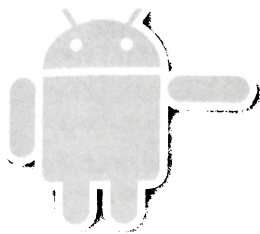
Nota: para testar o gesto de pinch/zoom, utilize um dispositivo real. Ou caso esteja utilizando o emulador do Genymotion mantenha a tecla Ctrl pressionada para fazer o zoom.

34.5 Links úteis

Neste capítulo, estudamos como incorporar o reconhecimento de gestos no aplicativo. Segue um link para você continuar seus estudos.

- **Android Training – Detecting Common Gestures**

<https://developer.android.com/training/gestures/detector.html>



CAPÍTULO 35

Sensores e Google Fit

Neste capítulo, vamos estudar como monitorar os sensores disponíveis no Android, como acelerômetro, giroscópio, temperatura, luminosidade, proximidade, entre outros, e, ainda, desenvolver vários aplicativos para demonstrar como utilizar cada sensor.

Também vamos estudar a plataforma do Google Fit que faz parte do Google Play Services.

35.1 Como obter a lista de sensores disponíveis

O Android tem três tipos de sensores: de movimento (acelerômetro e giroscópio), de posição (orientação, geomagnetismo, proximidade) e de ambiente (luminosidade, pressão e temperatura).

A lista de sensores disponíveis pode variar de acordo com cada aparelho, pois eles dependem do hardware. Quanto mais novo o aparelho, mais sensores ele terá. Os sensores também podem variar conforme a versão do Android, pois o suporte a novos sensores vai sendo adicionado à medida que o Android vai evoluindo. A figura 35.1, extraída da documentação oficial, mostra os sensores disponíveis para cada versão do Android, mas lembre-se de que, se o sensor vai existir ou não no aparelho, isso dependerá do hardware.

Fonte: http://developer.android.com/guide/topics/sensors/sensors_overview.html

Sensor	Android 4.0 (API Level 14)	Android 2.3 (API Level 9)	Android 2.2 (API Level 8)	Android 1.5 (API Level 3)
TYPE_ACCELEROMETER	Yes	Yes	Yes	Yes
TYPE_AMBIENT_TEMPERATURE	Yes	n/a	n/a	n/a
TYPE_GRAVITY	Yes	Yes	n/a	n/a
TYPE_GYROSCOPE	Yes	Yes	n/a ¹	n/a ¹
TYPE_LIGHT	Yes	Yes	Yes	Yes
TYPE_LINEAR_ACCELERATION	Yes	Yes	n/a	n/a
TYPE_MAGNETIC_FIELD	Yes	Yes	Yes	Yes
TYPE_ORIENTATION	Yes ²	Yes ²	Yes ²	Yes
TYPE_PRESSURE	Yes	Yes	n/a ¹	n/a ¹
TYPE_PROXIMITY	Yes	Yes	Yes	Yes
TYPE_RELATIVE_HUMIDITY	Yes	n/a	n/a	n/a
TYPE_ROTATION_VECTOR	Yes	Yes	n/a	n/a
TYPE_TEMPERATURE	Yes ²	Yes	Yes	Yes

Figura 35.1 – Lista de sensores por plataforma.

A lista a seguir explica os detalhes de cada sensor.

- **TYPE_ACCELEROMETER** – Informa a força da aceleração em m/s^2 aplicada no dispositivo nos eixos (x, y, e z), incluindo a força da gravidade.
- **TYPE_AMBIENT_TEMPERATURE** – Informa a temperatura ambiente em graus Celsius ($^{\circ}\text{C}$). Essa constante foi criada no Android 4.0 e substitui a constante **TYPE_TEMPERATURE**.
- **TYPE_GRAVITY** – Informa a força da gravidade em m/s^2 aplicada no dispositivo nos eixos (x, y, z).
- **TYPE_GYROSCOPE** – Informa as medidas da taxa de rotação em rad/s aplicada no dispositivo nos eixos (x, y, z).
- **TYPE_LIGHT** – Informa o nível de luz no ambiente em lux.
- **TYPE_LINEAR_ACCELERATION** – Informa a força da aceleração em m/s^2 aplicada no dispositivo em todos os eixos (x, y, e z), desconsiderando a força da gravidade.
- **TYPE_MAGNETIC_FIELD** – Mede a força do campo magnético ambiente aplicado no dispositivo nos eixos (x, y, e z) em μT (microteslas).
- **TYPE_PRESSURE** – Mede a pressão do ar ambiente em hPa ou mbar.
- **TYPE_PROXIMITY** – Mede a proximidade de um objeto em centímetros (cm) em relação à tela vista do dispositivo. Este sensor é usado pelo Android quando um usuário atende uma chamada telefônica, para desligar o LCD da tela se o aparelho estiver próximo da orelha do usuário.
- **TYPE_RELATIVE_HUMIDITY** – Mede a umidade do ambiente em percentagem (%).

- `TYPE_ROTATION_VECTOR` – Mede a orientação de um dispositivo fornecendo os três vetores de rotação.
- `TYPE_STEP_COUNTER` – Retorna a quantidade de passos dado pelo usuário desde que ligou o aparelho pela última vez.
- `TYPE_HEART_RATE` – Retorna a quantidade de batimentos cardíacos por minuto. Para utilizá-lo, é necessário declarar a permissão `android.permission.BODY_SENSORS`.

Para acessar um sensor, basta chamar o método `getDefaultSensor(tipo)` da classe `SensorManager` informando como parâmetro uma das constantes acima que acabamos de ver.

```
SensorManager sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
Sensor sensor = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
```

Atenção: o método `getDefaultSensor(tipo)` pode retornar nulo caso o sensor não exista no aparelho, portanto sempre verifique o retorno antes de utilizar o sensor.

O método `getDefaultSensor(tipo)` retorna o sensor padrão para o tipo especificado. Saiba que pode existir mais de um sensor do mesmo tipo, pois o aparelho pode embutir sensores de diferentes fabricantes.

Para começarmos a brincar com sensores, o primeiro passo é descobriremos quais sensores estão disponíveis em determinado dispositivo. Isso pode ser feito com o método `getSensorList(tipo)`, conforme demonstrado a seguir:

```
SensorManager sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
List<Sensor> sensorList = sensorManager.getSensorList(Sensor.TYPE_ALL);
```

O parâmetro `Sensor.TYPE_ALL` vai retornar todos os sensores disponíveis, mas também poderíamos ter utilizado uma constante específica de um tipo de sensor, como por exemplo: `Sensor.TYPE_ACCELEROMETER`, `Sensor.TYPE_GYROSCOPE`, `Sensor.TYPE_LIGHT` etc.

Para demonstrar como listar os sensores disponíveis no celular, vamos criar uma `activity` como esta. Caso prefira, abra o projeto **HelloSensores** no Android Studio e acompanhe a explicação.

ListaSensoresActivity.java

```
public class ListaSensoresActivity extends AppCompatActivity implements AdapterView.
OnClickListener {
    private SensorManager sensorManager;
    private List<Sensor> sensorList;
    @Override
```

```

public void onCreate(Bundle icle) {
    super.onCreate(icle);
    setContentView(R.layout.activity_lista_sensores);
    getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
    sensorList = sensorManager.getSensorList(Sensor.TYPE_ALL);
    List<String> nomes = new ArrayList<>();
    for (Sensor s : sensorList) {
        nomes.add(s.getName() + " - " + s.getVendor() + " - " + s.getType());
    }
    ListView listView = (ListView) findViewById(R.id.listView);
    listView.setOnItemClickListener(this);
    int layout = android.R.layout.simple_list_item_1;
    ArrayAdapter<String> adaptador = new ArrayAdapter<String>(this, layout, nomes);
    listView.setAdapter(adaptador);
}

@Override
public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
    Sensor sensor = sensorList.get(position);
    String msg = sensor.getName() + " - " + sensor.getVendor();
    Toast.makeText(this, msg, Toast.LENGTH_SHORT).show();
}
}

```

 /res/layout/activity_lista_sensores.xml

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout . . .>
    <ListView android:id="@+id/listView"
        android:layout_width="match_parent" android:layout_height="match_parent" />
</LinearLayout>

```

Ao executar esse exemplo, a lista de sensores disponíveis no aparelho será exibida na lista conforme a figura 35.2. O resultado desta figura são os sensores disponíveis no Nexus 5 com Android 5.1 (Lollipop). A figura demonstra a rolagem para baixo na lista para visualizar todos os sensores encontrados.

Nota: para testar esses exemplos, use um aparelho com Android, pois o emulador não consegue simular os sensores. A lista de sensores disponíveis vai variar de acordo com cada dispositivo, pois eles estão fortemente atrelados ao hardware. Quanto mais novo o aparelho, mais sensores ele terá.

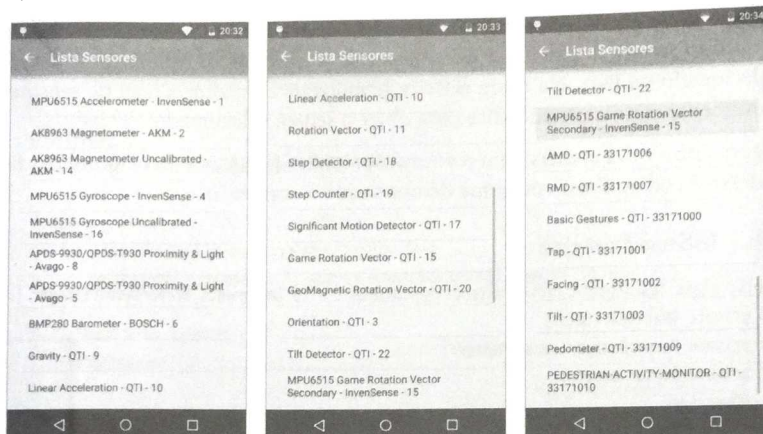


Figura 35.2 – Lista de sensores.

35.2 Testando os sensores

Para melhorar o exemplo anterior que mostra a lista dos sensores, vamos tratar o evento de clique na lista para permitir que o usuário selecione um sensor. Ao clicar na lista, vamos abrir uma nova activity, a qual vai ativar o sensor selecionado para ler os dados.

Altere o método `onItemClick(...)` que trata o evento de seleção na lista para navegar para a activity `TestSensorActivity`.



ListaSensoresActivity.java

```
public class ListaSensoresActivity extends AppCompatActivity implements AdapterView.
OnItemClickListener {
    @Override
    public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
        Sensor sensor = sensorList.get(position);
        // Navega para a activity que vai ligar este sensor
        Intent intent = new Intent(this, TestSensorActivity.class);
        intent.putExtra("position", position); // posição do sensor na lista
        startActivity(intent);
    }
}
```


Observe que não é possível passar o objeto `Sensor` como parâmetro, pois ele não é do tipo `Serializable` nem do tipo `Parcelable`, portanto estou passando o índice selecionado na lista. Na outra activity, teremos de consultar a lista de sensores novamente e utilizar este índice para obter o sensor selecionado.

Agora crie a nova activity com o wizard **New Activity > Blank Activity** do Android Studio e deixe o código-fonte conforme demonstrado a seguir:



TestSensorActivity.java

```
public class TestSensorActivity extends AppCompatActivity implements SensorEventListener {
    private TextView tValor;
    private SensorManager sensorManager;
    private Sensor sensor;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_test_sensor);
        // Posição do sensor
        int position = getIntent().getIntExtra("position", 0);
        // Busca o sensor pela posição
        sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
        List<Sensor> sensorList = sensorManager.getSensorList(Sensor.TYPE_ALL);
        sensor = sensorList.get(position);
        tValor = (TextView) findViewById(R.id.tValor);
        // Mostra o nome e fabricante do sensor
        getSupportActionBar().setTitle(sensor.getName());
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
    }
    @Override
    protected void onResume() {
        super.onResume();
        // Registra o sensor selecionado
        sensorManager.registerListener(this, sensor, SensorManager.SENSOR_DELAY_NORMAL);
    }
    @Override
    protected void onPause() {
        super.onPause();
        // Cancela o registro do sensor
        sensorManager.unregisterListener(this);
    }
}
```

```

@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
    // Chamado se mudou o status de precisão do sensor
}

@Override
public void onSensorChanged(SensorEvent event) {
    // Lê os valores do sensor
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < event.values.length; i++) {
        sb.append(i).append(": ").append(event.values[i]).append("\n");
    }
    // Mostra os valores
    tValor.setText(sb.toString());
}
}

```

No arquivo de layout, vamos ter um `TextView` para mostrar os valores do sensor. O sensor retorna os dados no formato de um array do tipo `float`, portanto pode ter mais de um valor dependendo do tipo do sensor.



/res/layout/activity_test_sensor.xml

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout ... android:orientation="vertical" >
    <TextView
        android:id="@+id/tValor" android:text="16sp"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
</LinearLayout>

```

No código da `activity`, logo depois de obter o sensor selecionado, é iniciado o monitoramento chamando o método `registerListener(listener, sensor, rate)`, informando os seguintes argumentos:

Parâmetro	Descrição
<code>SensorEventListener listener</code>	Interface para receber os valores do sensor.
<code>Sensor sensor</code>	Sensor selecionado.
<code>int rateUs</code>	Intervalo desejado para receber os eventos do sensor.

O parâmetro `rateUs` do método `registerListener(listener, sensor, rate)` pode receber alguma destas constantes, as quais definem o intervalo (velocidade) com que o sensor vai enviar informações para o aplicativo:

Constante	Descrição
SENSOR_DELAY_NORMAL	Intervalo padrão para enviar os valores do sensor a cada 200.000 microssegundos.
SENSOR_DELAY_UI	Intervalo de 60.000 microssegundos.
SENSOR_DELAY_GAME	Intervalo de 20.000 microssegundos.
SENSOR_DELAY_FASTEST	Esta constante envia os valores do sensor o mais rápido possível sem nenhum atraso.

Depois de registrar o listener, é necessário implementar a interface `SensorEventListener` que contém os métodos `onSensorChanged(event)` e `onAccuracyChanged(precisao)`. O método `onSensorChanged(event)` é chamado sempre que o sensor tiver informações, conforme o intervalo (rate) registrado no listener. Neste método, a aplicação deve ler os valores do sensor, os quais são sempre retornados como um array de `float`. No caso dos sensores de acelerômetro e giroscópio, são retornadas três posições no array, referentes aos eixos x, y e z.

```
public void onSensorChanged(SensorEvent event) {  
    // Lê os valores do sensor  
    float x = event.values[0];  
    float y = event.values[1];  
    float z = event.values[2];  
}
```

Mas outros sensores como luminosidade, pressão e proximidade retornam apenas um valor, portanto podemos ler somente a posição 0 (zero) do array.

```
public void onSensorChanged(SensorEvent event) {  
    // Lê o valor do sensor  
    float valor = event.values[0];  
}
```

Se estiver na dúvida de como ler os valores, é simples: basta verificar o tamanho do array.

O método `onAccuracyChanged(precisao)` é chamado para informar que a precisão do sensor mudou e o parâmetro `accuracy` representa algum dos seguintes status: `SENSOR_STATUS_ACCURACY_LOW` para sinal baixo, `SENSOR_STATUS_ACCURACY_MEDIUM` para sinal médio, `SENSOR_STATUS_ACCURACY_HIGH` para sinal alto e `SENSOR_STATUS_UNRELIABLE` para sinal indisponível. Em uma aplicação real, devemos monitorar esses status para verificar a qualidade do sinal do sensor.

Algo importante sobre os sensores é o tratamento do ciclo de vida da aplicação. O monitoramento do sensor deve ser iniciado no método `onResume()` e parado no método `onPause()`. Acredito que a esta altura do livro você já saiba por que isso é assim.

Capítulo 35 ■ Sensores e Google Fit

```
protected void onResume() {
    super.onResume();
    // Registra o sensor selecionado
    sensorManager.registerListener(this, sensor, SensorManager.SENSOR_DELAY_NORMAL);
}

@Override
protected void onPause() {
    super.onPause();
    // Cancela o registro do sensor
    sensorManager.unregisterListener(this);
}
```

Para testar este exemplo, execute o projeto em um dispositivo real, depois clique em algum sensor da lista. A figura 35.3 mostra o resultado do sensor de acelerômetro. Veja que no array existem três posições, que são referentes aos eixos x, y e z do acelerômetro.

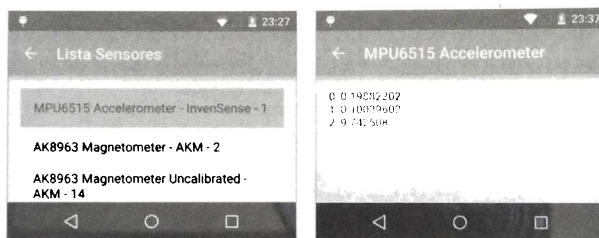


Figura 35.3 – Sensor de acelerômetro.

A figura 35.4 mostra o resultado ao clicar no sensor do contador de passos. Neste caso o array contém apenas uma posição, que é a quantidade de passos. Aproveitando, caso precise desenvolver algo relacionado a contador de passos ou de batimentos cardíacos, recomendo estudar o aplicativo do Google Fit e a Fitness API. No final do capítulo faremos um exemplo com a Fitness API.

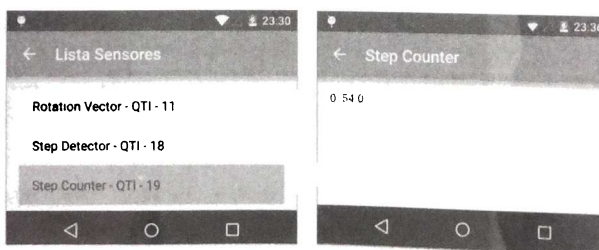


Figura 35.4 – Sensor de contador de passos.

35.3 Sensor de luminosidade

Embora o exemplo anterior já tenha sido suficiente para mostrar como listar todos os sensores e ainda obter o valor de cada sensor, vamos explorar alguns sensores individuais apenas para praticar.

O sensor de luminosidade é representado pela constante `Sensor.TYPE_LIGHT` e informa a intensidade da luz e claridade no ambiente externo. Nesse sensor, para ler o valor da intensidade da luz, basta acessar a primeira posição do array, em que o número float representa o valor obtido do sensor.

```
public void onSensorChanged(SensorEvent event) {
    // Lê o valor do sensor
    float valor = event.values[0];
}
```

O intervalo de números que o sensor vai retornar variará conforme o fabricante. Para descobrir o valor máximo retornado pelo sensor, utilize o método `sensor.getMaximumRange()`. Sendo assim, o sensor pode retornar valores entre 0 (zero) e o valor máximo. Justamente por isso, para demonstrar o sensor de luminosidade, vamos criar um arquivo de layout com a view `SeekBar`.



/res/layout/activity_sensor_seekbar.xml

```
<LinearLayout . . . android:orientation="vertical">
    <TextView android:id="@+id/tValor"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <SeekBar android:id="@+id/progress"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="fill_parent" android:layout_height="wrap_content"
        android:padding="20dp"
        android:progress="0" android:max="100" />
</LinearLayout>
```

Agora, vamos escrever o código da activity, que vai utilizar o sensor do tipo luminosidade com a constante `Sensor.TYPE_LIGHT`. Veja que estamos obtendo o sensor com o método `sensorManager.getDefaultSensor(tipo)`; dessa maneira, é possível obter o sensor padrão caso exista mais de um. De qualquer forma, é importante validar se o sensor existe, pois, como já foi explicado, tudo depende do hardware do aparelho.



LuminosidadeActivity.java

```
public class LuminosidadeActivity extends AppCompatActivity implements SensorEventListener {
    private static final int TIPO_SENSOR = Sensor.TYPE_LIGHT;
    private SensorManager sensorManager;
```

Capítulo 35 ■ Sensores e Google Fit

```

private Sensor sensor;
private SeekBar progress;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_sensor_seekbar);
    progress = (SeekBar) findViewById(R.id.progress);
    sensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);
    sensor = sensorManager.getDefaultSensor(TIPO_SENSOR);
    if (sensor != null) {
        // Define o valor máximo do SeekBar
        float max = sensor.getMaximumRange();
        progress.setMax((int) max);
    } else {
        Toast.makeText(this, "Sensor não disponível", Toast.LENGTH_SHORT).show();
    }
}
@Override
protected void onResume() {
    super.onResume();
    if (sensor != null) {
        sensorManager.registerListener(this, sensor, SensorManager.SENSOR_DELAY_NORMAL);
    }
}
@Override
protected void onPause() {
    super.onPause();
    sensorManager.unregisterListener(this);
}
@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
    // Mudou o status de precisão do sensor
}
@Override
public void onSensorChanged(SensorEvent event) {
    // Lê o valor do sensor (intensidade da luz)
    float valor = event.values[0];
    ((TextView) findViewById(R.id.tValor)).setText("Luz: " + valor);
    // Atualiza o valor no SeekBar
    progress.setProgress((int) valor);
}
}

```

Caso você tenha um aparelho com o sensor de luminosidade, brinque com os resultados testando o sensor em ambientes com bastante luz ou escuros. Você verá que a *SeekBar* vai mudar constantemente, pois está medindo a intensidade da luz no local.

A figura 35.5 mostra o resultado. No lado esquerdo, coloquei o aparelho embaixo de uma mesa de forma a pegar pouca luz. No lado direito, coloquei o aparelho direto no sol, então note que a *SeekBar* está totalmente preenchida.

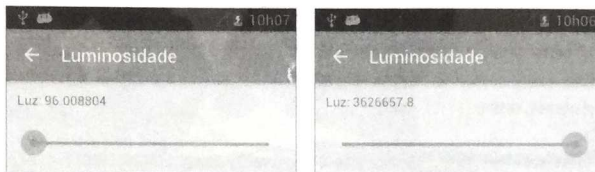


Figura 35.5 – Sensor de luminosidade.

35.4 Sensor de temperatura

O sensor de temperatura é representado pela constante `Sensor.TYPE_AMBIENT_TEMPERATURE` e informa a temperatura em graus Celsius (°C). Para utilizar esse sensor, o procedimento é exatamente igual ao do exemplo anterior. O valor retornado será a temperatura externa.

```
public void onSensorChanged(SensorEvent event) {  
    // Valor da temperatura em °C  
    float valor = event.values[0];  
}
```

É importante ressaltar que antigamente era utilizada a constante `Sensor.TYPE_TEMPERATURE` para esse sensor, mas a partir do Android 4.0 (API Level 14) essa constante foi descontinuada (deprecated) e, agora, a constante `Sensor.TYPE_AMBIENT_TEMPERATURE` deve ser utilizada.

Como a utilização desse sensor é idêntica à do exemplo anterior, não vamos demonstrar nenhum código.

35.5 Sensor de proximidade

O sensor de proximidade é representado pela constante `Sensor.TYPE_PROXIMITY` e possibilita verificar o quão próximo o aparelho está de determinado objeto.

Capítulo 35 ■ Sensores e Google Fit

Esse sensor é utilizado diretamente no Android pela aplicação que controla as ligações telefônicas, pois, sempre que atendemos uma ligação e aproximamos o celular do ouvido, ele detecta a proximidade e desliga o LCD da tela, para economizar recursos durante a ligação. Quando terminamos de conversar, ao afastarmos o celular do ouvido, automaticamente o LCD da tela é ligado.

Para utilizar esse sensor, o procedimento é exatamente igual ao do exemplo da luminosidade ou temperatura, e o sensor retorna apenas um valor, que é a distância em centímetros do objeto que está próximo do celular. Portanto, para ler esse valor, podemos utilizar o seguinte código:

```
public void onSensorChanged(SensorEvent event) {  
    // Distância em centímetros  
    float valor = event.values[0];  
}
```

Nesse sensor, também é interessante obter o valor máximo que ele pode retornar com o método `sensor.getMaximumRange()`, pois assim sabemos quais os valores que o sensor retorna entre 0 (zero) e o valor máximo. Geralmente o sensor retorna valores de 0 a 5 centímetros de proximidade, mas isso pode variar, dependendo do hardware.

35.6 Sensor de acelerômetro

O sensor de acelerômetro é representado pela constante `Sensor.TYPE_ACCELEROMETER` e permite monitorar a aceleração aplicada ao aparelho nos eixos x, y e z.

Esse é um dos sensores mais conhecidos, pois é frequentemente utilizado em jogos, mas também é um dos mais complicados de entender, porque é fundamental compreender bem o sistema de coordenadas em que ele atua, conforme a figura 35.6.

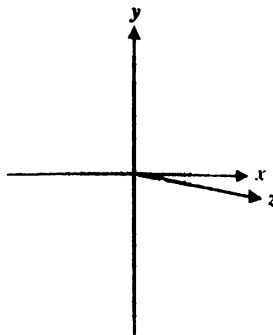


Figura 35.6 – Sistema de coordenadas do sensor.

O valor desse sensor varia entre 0 e aproximadamente 981, medido em m/s^2 .

Com base nessa figura, com o celular posicionado na vertical, podemos assumir que o valor do eixo Y é aproximadamente +981. Caso você gire o celular nesse eixo e o deixe de ponta-cabeça, o valor de y será -981 aproximadamente.

Agora, se você girar o celular para a direita, para deixá-lo na horizontal, o valor no eixo X será aproximadamente +981. Caso você gire o celular para a esquerda, deixando-o também na horizontal, o valor do eixo X será aproximadamente -981.

O eixo Z fica com o valor positivo, próximo de +981, quando o celular está deitado sobre uma mesa, de forma que não há forças exercendo pressão nos eixos x e y.

Com base nessas informações, é interessante criarmos um exemplo para você visualizar os valores retornados desse sensor, porque logo vamos ter de discutir alguns comportamentos que esse sensor apresenta ao girar o celular.

Vamos criar a activity `AcelerometroActivity` com o seguinte arquivo de layout:

```
 /res/layout/activity_sensor_acelerometro.xml

<LinearLayout . . . android:orientation="vertical" >
    <TextView android:id="@+id/tX" android:text="@string/x"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <TextView android:id="@+id/tY" android:text="@string/y"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <TextView android:id="@+id/tZ" android:text="@string/z"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
</LinearLayout>
```

Esse layout tem três `TextViews` para exibir os valores dos eixos x, y e z. Já o código da activity funciona da mesma forma que os outros sensores. Basta ler os dados do sensor do tipo `Sensor.TYPE_ACCELEROMETER`:

```
 AcelerometroActivity.java

public class AcelerometroActivity extends AppCompatActivity implements SensorEventListener {
    private static final int TIPO_SENSOR = Sensor.TYPE_ACCELEROMETER;
    . . .
    // Faça tudo igual aos outros exemplos aqui...
    // Basta iniciar o sensor, os resultados serão lidos no onSensorChanged(event)
    @Override
    public void onSensorChanged(SensorEvent event) {
        float sensorX = -event.values[0];
        float sensorY = event.values[1];
        float sensorZ = event.values[2];
```

Capítulo 35 ■ Sensores e Google Fit

```

    TextView tx = (TextView) findViewById(R.id.tX);
    TextView ty = (TextView) findViewById(R.id.tY);
    TextView tz = (TextView) findViewById(R.id.tZ);
    if (tx != null) {
        tx.setText("X: " + sensorX);
        ty.setText("Y: " + sensorY);
        tz.setText("Z: " + sensorZ);
    }
}
}

```

Para testar o acelerômetro, vamos travar a orientação da activity na vertical. Faça isso adicionando o atributo `android:screenOrientation="portrait"` ao arquivo de manifesto. Isso é necessário porque, caso a orientação do celular mude para a horizontal, os valores do acelerômetro mudarão também. Logo explicarei mais detalhes sobre isso.

```
<activity android:name=".AcelerometroActivity" android:screenOrientation="portrait" />
```

Agora, execute essa activity. A figura 35.7 mostra o resultado.

A primeira parte da figura mostra o resultado com o celular na vertical. Neste caso o eixo que sofre a maior alteração é o y, que mede aproximadamente +981 nessa posição.

A figura do meio mostra o resultado ao girar o celular para a direita, até deixá-lo na horizontal, de forma que os botões de **Home**, **Voltar** e **Menu** do Android fiquem à esquerda da tela. Nessa posição, o eixo que sofre alteração é o x.

Na parte da direita da figura, o celular está no chão ou sobre a mesa. Dessa vez, o eixo z sofrerá a alteração.

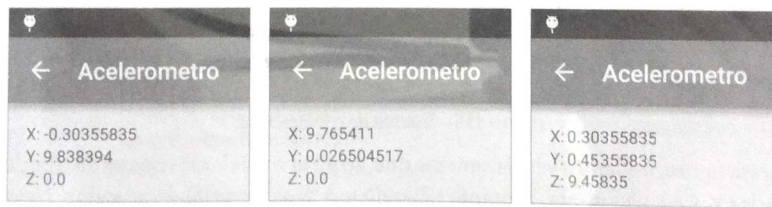


Figura 35.7 – Valores do acelerômetro nos eixos x, y e z.

Até aqui, transcorreu tudo bem. Os valores retornados pelo sensor do acelerômetro foram exatamente os esperados. Contudo, vamos remover a configuração da activity que estava travando a tela na vertical para permitir que o usuário gire o celular. Portanto, deixe o arquivo *AndroidManifest.xml* da seguinte forma:

```
<activity android:name=".AcelerometroActivity" />
```

Feito isso, vamos executar novamente o exemplo e conferir o resultado.

Ao deixar o celular na vertical, tudo funciona normalmente, pois essa é a orientação padrão dos smartphones. Portanto, não há novidades no eixo y .

O problema está no eixo x , pois ele reflete os valores na horizontal. Dessa forma, gire a tela e confira o resultado dos valores retornados pelo acelerômetro. O resultado é que os valores de x e y na horizontal estão invertidos.

Para auxiliar seu entendimento, é só você imaginar que está jogando um jogo como o clássico Labirinto. Nele, você deve mover uma bola pelo caminho para encontrar a saída. Nessa posição, com o celular deitado na horizontal, para jogar geralmente você segura o celular com o visor do LCD para cima. Com o celular nessa posição, o que esperamos é que, ao girar o celular para a esquerda e direita, o eixo que sofrerá a alteração seja o x ; e, ao mover para cima e para baixo, seja o y . Mas se o celular estiver deitado, não é isso o que acontecerá, porque, não importa a posição do celular, o sistema de coordenadas do acelerômetro nunca mudará.

Para entender melhor essa situação, veja os eixos x , y e z do sistema de coordenadas, com o celular na vertical e horizontal, conforme demonstrado na figura 35.8:

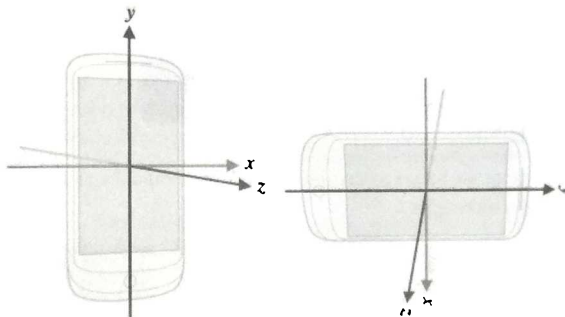


Figura 35.8– Sistema de coordenadas.

Nessa figura, podemos ver claramente que, ao girar o celular, o sistema de coordenadas x , y e z não muda, portanto não reflete a posição atual do aparelho. Nesse caso, o que faremos é compensar essa rotação, pois é possível identificar em qual posição o celular se encontra e trocar os valores dos eixos x e y retornados pelo acelerômetro. Para descobrir a orientação atual do dispositivo, podemos utilizar a classe `Display` e o método `getRotation()`:

```
WindowManager windowManager = (WindowManager) getSystemService(WINDOW_SERVICE);  
Display display = windowManager.getDefaultDisplay();
```

Capítulo 35 ■ Sensores e Google Fit

```
// Para descobrir a rotação do aparelho
switch (display.getRotation()) {
    case Surface.ROTATION_0:
        // Vertical
    case Surface.ROTATION_90:
        // Horizontal (botões para a direita)
    case Surface.ROTATION_180:
        // Vertical: de ponta-cabeça
    case Surface.ROTATION_270:
        // Horizontal (botões para a esquerda)
}
```

Esse código permite descobrir a rotação e compensar os valores de x e y, que, na prática, serão trocados, pois a rotação do celular está diferente do sistema de coordenadas utilizado pelo acelerômetro, visto que esse sistema nunca muda. Para organizar o projeto, deixaremos este trecho de código na classe `SensorUtil`:



`SensorUtil.java`

```
...
public class SensorUtil {
    public static float[] fixAcelerometro(Context context, SensorEvent event) {
        float sensorX = 0;
        float sensorY = 0;
        float sensorZ = 0;
        WindowManager windowManager = (WindowManager)
            context.getSystemService(Context.WINDOW_SERVICE);
        Display display = windowManager.getDefaultDisplay();
        // Verifica a rotação do aparelho
        switch (display.getRotation()) {
            case Surface.ROTATION_0:
                // Vertical
                sensorX = -event.values[0];
                sensorY = event.values[1];
                break;
            case Surface.ROTATION_90:
                // Horizontal (botões para a direita)
                sensorX = event.values[1];
                sensorY = event.values[0];
                break;
            case Surface.ROTATION_180:
                // Vertical: de ponta-cabeça
                sensorX = -event.values[0];
```

```

        sensorY = -event.values[1];
        break;
    case Surface.ROTATION_270:
        // Horizontal (botões para a esquerda)
        sensorX = -event.values[1];
        sensorY = -event.values[0];
        break;
    }
    // Troca os valores de x e y
    float[] values = new float[3];
    values[0] = sensorX;
    values[1] = sensorY;
    values[2] = sensorZ;
    return values;
}
}

public static String getRotationString(Context context) {
    WindowManager windowManager = (WindowManager)
        context.getSystemService(Context.WINDOW_SERVICE);
    Display display = windowManager.getDefaultDisplay();
    // Verifica a rotação do aparelho
    switch (display.getRotation()) {
        case Surface.ROTATION_0:
            return "ROTATION_0";
        case Surface.ROTATION_90:
            return "ROTATION_90";
        case Surface.ROTATION_180:
            return "ROTATION_180";
        case Surface.ROTATION_270:
            return "ROTATION_270";
    }
    return null;
}
}
}

```

Agora, podemos atualizar o código da activity para corrigir os valores do sensor, conforme a rotação do aparelho:



AcelerometroActivity.java

```

public class AcelerometroActivity extends AppCompatActivity implements SensorEventListener {
    private long time;
    . . .
    @Override
    public void onSensorChanged(SensorEvent event) {

```

```

    long now = System.currentTimeMillis();
    if(now - time > 1000) {
        time = now;
        float values[] = SensorUtil.fixAcelerometro(this, event);
        float sensorX = values[0];
        float sensorY = values[1];
        float sensorZ = values[2];
        TextView tx = (TextView) findViewById(R.id.tx);
        TextView ty = (TextView) findViewById(R.id.ty);
        TextView tz = (TextView) findViewById(R.id.tz);
        TextView tMsg = (TextView) findViewById(R.id.tMsg);
        if (tx != null) {
            if(tx != null) { tx.setText("X: " + sensorX); }
            if(ty != null) { ty.setText("Y: " + sensorY); }
            if(tz != null) { tz.setText("Z: " + sensorZ); }
            if(tMsg != null) { tMsg.setText("Rotação: "
                + SensorUtil.getRotationString(this)); }
        }
    }
}
}
}

```

Nesse exemplo, para efeito de testes, foi inserido mais um `TextView` com o identificador `@id/tMsg` no arquivo de layout, para mostrarmos o retorno do método `SensorUtil.getRotationString(context)`.

 /res/layout/activity_sensor_acelerometro.xml

```

<LinearLayout . . . android:orientation="vertical" >
    <TextView android:id="@id/tx" . . . />
    <TextView android:id="@id/ty" . . . />
    <TextView android:id="@id/tz" . . . />
    <TextView android:id="@id/tMsg"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
</LinearLayout>

```

Feito isso, execute novamente o exemplo para conferir o resultado. É interessante que você gire o celular e entenda os resultados retornados pelo sensor. Na verdade, o que a classe `SensorUtil` faz é inverter os eixos x e y caso o celular esteja na horizontal.

Nota: é importante ressaltar que a orientação padrão de um aparelho pode não ser sempre vertical. Por exemplo, nos tablets, a orientação padrão geralmente é horizontal. Nesse caso, quando a rotação indicar `ROTATION_0`, significará que ele está na horizontal, isto é, em sua posição padrão. Portanto, lembre-se de que

essas constantes de rotação indicam se o aparelho teve sua tela rotacionada, a partir de sua posição padrão, que pode ser vertical nos smartphones ou horizontal nos tablets.

35.7 Movendo uma view pela tela com o acelerômetro

O sensor do acelerômetro é muito utilizado em jogos. Vamos brincar um pouco apenas para ter uma ideia de como podemos movimentar uma view pela tela utilizando o acelerômetro.

Vamos criar uma view customizada que vai desenhar uma imagem de um boneco do Android. A imagem será desenhada nas posições x e y, conforme os valores dos atributos dx e dy definidos na classe. Na activity vamos ler os valores do sensor do acelerômetro e atualizar os atributos dx e dy desta view, assim obteremos o efeito de o boneco do Android ficar se movendo pela tela.



BonecoAndroidView.java

```
package br.com.livroandroid.sensores;

...
public class BonecoAndroidView extends View {
    private Paint paint = new Paint();
    private Drawable drawable;
    private int dx, dy;
    public BonecoAndroidView(Context context, AttributeSet attrs) {
        super(context, attrs);
        init(context);
    }
    public BonecoAndroidView(Context context) {
        super(context);
        init(context);
    }
    private void init(Context context) {
        // Configura o fundo cinza e cria a imagem
        drawable = context.getResources().getDrawable(R.drawable.android);
        drawable.setBounds(0, 0, drawable.getIntrinsicWidth(),
            drawable.getIntrinsicHeight());
    }
    @Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
```

Capítulo 35 ■ Sensores e Google Fit

```

        paint.setColor(Color.LTGRAY);
        // Desenha o fundo da view (um quadrado cinza)
        canvas.drawRect(0, 0, getWidth(), getHeight(), paint);
        paint.setTextSize(toPixels(16));
        paint.setColor(Color.BLACK);
        canvas.drawText("x/y:" + dx + "/" + dy, 50, 50, paint);
        // Desenha a imagem das posições x e y
        canvas.translate(dx, dy);
        drawable.draw(canvas);
    }

    private float toPixels(int dp) {
        final float scale = getResources().getDisplayMetrics().density;
        float px = dp * scale + 0.5f;
        return px;
    }

    public void setDx(int dx) { this.dx = dx; }
    public void setDy(int dy) { this.dy = dy; }
    public int getDy() { return dy; }
    public int getDx() { return dx; }
    public Drawable getDrawable() { return drawable; }
}

```

Na sequência, crie a classe `AcelerometroJogoActivity`. Ela vai herdar de `AcelerometroActivity` que criamos anteriormente, para aproveitarmos o código que ativa o sensor. Assim, ela ficará apenas com a lógica de movimentar a view.

`AcelerometroJogoActivity.java`

```

...
public class AcelerometroJogoActivity extends AcelerometroActivity {
    // Posições para desenhar a imagem
    private BonecoAndroidView androidView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_acelerometro_jogo);
        androidView = (BonecoAndroidView) findViewById(R.id.bonecoView);
    }

    @Override
    public void onSensorChanged(SensorEvent event) {
        super.onSensorChanged(event);
        // Lê os valores retornados pelo acelerômetro
        float values[] = SensorUtil.fixAcelerometro(this, event);
        float sensorX = values[0];
    }
}

```



```

float sensorY = values[1];
// Vai incrementando os valores de x e y para o objeto se mover
int newdx = androidView.getDx() + (int) sensorX * 10;
int newdy = androidView.getDy() + (int) sensorY * 10;
int imgW = androidView.getDrawable().getIntrinsicWidth();
int imgH = androidView.getDrawable().getIntrinsicHeight();
// Não deixa o valor ficar negativo ou maior que o tamanho da tela
DisplayMetrics displayMetrics = getResources().getDisplayMetrics();
if (!(newdx < 0 || newdx + imgW > displayMetrics.widthPixels)) {
    // Atualiza o eixo x
    androidView.setDx(newdx);
}
int actionBarH = displayMetrics.heightPixels - androidView.getHeight();
if (!(newdy < 0 || newdy + imgH > displayMetrics.heightPixels - actionBarH)) {
    // Atualiza o eixo y
    androidView.setDy(newdy);
}
// Redesenha a view
androidView.invalidate();
}
}

```

No layout da activity, vamos colocar um `TextView` para mostrar os valores dos eixos `x` e `y`. Também vamos adicionar a view customizada que vai desenhar o boneco do Android.



/res/layout/activity_acelerometro_jogo.xml

```

<LinearLayout . . . android:orientation="vertical">
    <TextView android:id="@+id/tx"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/x" android:textSize="18sp" />
    <TextView android:id="@+id/ty"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/y" android:textSize="18sp" />
    <br.com.livroandroid.sensores.BonecoAndroidView android:id="@+id/bonecoView"
        android:layout_width="match_parent" android:layout_height="0dp"
        android:layout_weight="9" />
</LinearLayout>

```

Ao executar esse exemplo, você poderá mover o bonequinho do Android pela tela como se estivesse jogando. A figura 35.9 demonstra o resultado quando o boneco está praticamente no centro da tela. A implementação é simples. Estamos monitorando o sensor do acelerômetro e atualizando a propriedade `dx` e `dy` da view

customizada. Com isso a view vai desenhar a imagem nessa posição. No layout mostramos os valores de x e y do acelerômetro. Dentro da view é desenhado um quadrado cinza e a imagem do Android que fica se movendo. No topo esquerdo do quadrado, é desenhada a posição dx e dy na qual o boneco do Android foi desenhado. Acredito que se você executar o exemplo no emulador ficará mais simples de entender; portanto, faça isso. Note que este exemplo é simples e não tratou aspectos mais avançados como aceleração e gravidade, mas o objetivo foi apenas brincar um pouco.

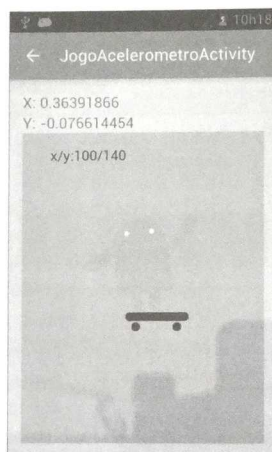


Figura 35.9 – Movendo um objeto pela tela com o acelerômetro.

35.8 Google Fit

O Google Fit é uma plataforma completa que permite aos desenvolvedores obter os dados dos sensores e ainda salvar ou ler os dados na nuvem do Google. Inclusive existe o aplicativo Google Fit para celulares, tablets e wearables. Também existe o site <https://fit.google.com/> para visualizarmos os dados salvos pelo aplicativo padrão do Google Fit.

O melhor de tudo é que essas APIs estão disponíveis para o desenvolvedor; logo, caso você esteja desenvolvendo aplicativos para a área de saúde que precisam utilizar o contador de passos e batimentos cardíacos, as APIs do Google Fit podem ajudá-lo.

O Google Fit faz parte do Google Play Services, então para utilizar esses serviços precisamos nos conectar aos servidores do Google, conforme aprendemos no

capítulo 23, sobre localização e GPS. Isso é feito porque os dados dos sensores podem ficar salvos na nuvem do Google, conforme ilustrado na figura 35.10.

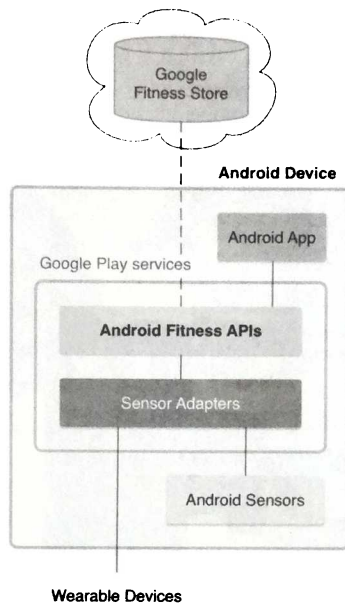


Figura 35.10 – Google Fit.

As APIs do Google Fit, também conhecidas como Fitness API, são compostas de outras subAPIs, cujas principais estão listadas a seguir:

- **SensorsApi** – Esta é a API mais utilizada e permite ler os dados dos sensores de contador de passos e batimentos cardíacos de forma simples.
- **RecordingApi** – Permite coletar os dados e gravar no Google Fit.
- **SessionsApi** – Permite ao aplicativo gerenciar sessões de atividades do usuário.
- **HistoryApi** – Permite consultar os dados inseridos na base do Google Fit.

Agora vamos ver um pouco de código para criar um exemplo de pedômetro, ou seja, um aplicativo que conta os passos do usuário. Tudo começa criando um objeto do tipo `GoogleApiClient` para se conectar ao Google Play Services. Ao fazer isso, é adicionada a Fitness API no escopo da classe `GoogleApiClient`.

Capítulo 35 ■ Sensores e Google Fit

```
// Cria o cliente para se conectar ao Google Play Services
GoogleApiClient mGoogleApiClient = new GoogleApiClient.Builder(this)
    .addApi(Fitness.SENSORS_API) // Adiciona a Fitness API
    .addScope(new Scope(Scopes.FITNESS_LOCATION_READ))
    .addScope(new Scope(Scopes.FITNESS_ACTIVITY_READ))
    .addOnConnectionsCallbacks(this)
    .addOnConnectionFailedListener(this)
    .build();
mGoogleApiClient.connect();
```

No capítulo 23, sobre localização, estudamos como se conectar ao Google Play Services, portanto vamos continuar com as partes referentes ao Google Fit. Quando a conexão com o servidor do Google for realizada, o método `onConnected(bundle)` será chamado e neste momento podemos iniciar a API do Google Fit. O trecho de código a seguir mostra como iniciar o contador de passos.

```
@Override
public void onConnected(Bundle connectionHint) {
    // Conectou-se ao Google Play Services
    // Então podemos nos conectar ao Google Fit
    // Contador de passos (TYPE_STEP_COUNT_DELTA)
    SensorRequest req = new SensorRequest.Builder()
        .setDataType(DataType.TYPE_STEP_COUNT_DELTA)
        .setSamplingRate(1, TimeUnit.SECONDS).build();
    // Ativa o contador de passos
    Fitness.SensorsApi.add(mGoogleApiClient, request, this);
}
```

Feito isso, sua classe terá de implementar a interface `OnDataPointListener` do Google Fit, e os resultados do sensor serão entregues no método `onDataPoint(dataPoint)`.

```
@Override
public void onDataPoint(DataPoint dataPoint) {
    // Ler os dados do pedômetro aqui...
}
```

Pronto! No método `onDataPoint(dataPoint)` basta ler os passos, que serão entregues automaticamente pela API do Google Fit, a qual é compatível com o Android 2.3 (API Level 9). Sua vantagem é que ela faz parte de toda uma plataforma, inclusive facilitando persistir todos os dados e consultá-los na nuvem do Google.

Um aspecto negativo é que para o Google Fit funcionar é necessária uma conexão com a internet para pelo menos se conectar ao Google Play Services. Depois que o aplicativo fez a conexão pela primeira vez, ele pode trabalhar de forma offline, mas de qualquer forma é necessária essa conexão inicial. Utilizando o Google Fit,

os dados serão salvos na nuvem do Google, e depois com a **HistoryApi** é possível inclusive consultar esses dados.

Caso seja um requisito do seu projeto que o aplicativo deva funcionar totalmente offline, não é possível utilizar o Google Fit. Nesse caso deve-se utilizar o sensor do tipo `TYPE_STEP_COUNTER`, conforme estudamos neste capítulo. A desvantagem desta solução é que se perde o poder de toda a plataforma do Google Fit.

Mas chega de conversa! Vamos criar um exemplo de aplicativo que utiliza o Google Fit para criar um pedômetro.

Primeiramente, declare a dependência do Google Play Services:



app/build.gradle

. . .

```
compile 'com.google.android.gms:play-services:7.0.0'
```

Na sequência, crie a activity com o seguinte código. Basicamente ela se conecta ao Google Play Services como já fizemos no exemplo de localização, e no método `onConnected(bundle)` é iniciada a conexão com o Google Fit para contar os passos.



GoogleFitPedometroActivity.java

```
public class GoogleFitPedometroActivity extends AppCompatActivity implements
GoogleApiClient.ConnectionCallbacks, GoogleApiClient.OnConnectionFailedListener {
    private static final String TAG = "livroandroid";
    private GoogleApiClient mGoogleApiClient;
    private TextView text;
    private int qtdePassos;
    private static final int REQUEST_OAUTH = 1;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_google_fit_pedometro);
        text = (TextView) findViewById(R.id.text);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        // Configura o objeto GoogleApiClient
        mGoogleApiClient = new GoogleApiClient.Builder(this)
            .addApi(Fitness.SENSORS_API)
            .addScope(new Scope(Scopes.FITNESS_LOCATION_READ))
            .addScope(new Scope(Scopes.FITNESS_ACTIVITY_READ))
            .addConnectionCallbacks(this)
            .addOnConnectionFailedListener(this)
            .build();
```

Capítulo 35 ■ Sensores e Google Fit

```

    }
    @Override
    protected void onStart() {
        super.onStart();
        // Conecta ao Google Play Services
        if(!mGoogleApiClient.isConnecting() && !mGoogleApiClient.isConnected()) {
            toast("mGoogleApiClient.connect()");
            mGoogleApiClient.connect();
        }
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // Desconecta ao sair
        mGoogleApiClient.disconnect();
    }
    @Override
    public void onConnected(Bundle bundle) {
        toast("Conectado ao Google Play Services!");
        startPedometer();
    }
    private void startPedometer() {
        Log.d("livroandroid", "startPedometer");
        // Listener do Fitness API que conta os passos
        OnDataPointListener listener = new OnDataPointListener() {
            @Override
            public void onDataPoint(DataPoint dataPoint) {
                for (Field field : dataPoint.getDataType().getFields()) {
                    if (dataPoint.getDataType().equals(DataType.TYPE_STEP_COUNT_DELTA)) {
                        Value val = dataPoint.getValue(field);
                        Log.d("livroandroid", "Valor Pedometro: " + val);
                        qtdePassos += val.asInt();
                        runOnUiThread(new Runnable() {
                            @Override
                            public void run() {
                                text.setText("Passos: " + qtdePassos);
                            }
                        });
                    }
                }
            }
        }
    }
};

```

```

        // Contador de passos (TYPE_STEP_COUNT_DELTA)
        SensorRequest req = new SensorRequest.Builder()
            .setDataType(DataType.TYPE_STEP_COUNT_DELTA)
            .setSamplingRate(1, TimeUnit.SECONDS)
            .build();

        // Ativa a API do Fitness
        PendingResult<Status> result = Fitness.SensorsApi.add(mGoogleApiClient, req, listener);
        Log.d("livroandroid", "Google Fit pedometer ativado: " + result);
    }

    @Override
    public void onConnectionSuspended(int cause) {
        toast("Conexão interrompida.");
    }

    @Override
    public void onConnectionFailed(ConnectionResult result) {
        if (!result.hasResolution()) {
            // Se for algum erro de configuração ou serviço, mostra alerta
            GooglePlayServicesUtil.getErrorDialog(result.getErrorCode(), this, 0).show();
            return;
        }
        // Caso contrário, pode ser porque o usuário não autorizou o acesso
        try {
            result.startResolutionForResult(this, REQUEST_OAUTH);
        } catch (IntentSender.SendIntentException e) {
            Log.e(TAG, "Erro ao autorizar: " + e.getMessage(), e);
        }
    }

    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
        if (requestCode == REQUEST_OAUTH) {
            if (resultCode == RESULT_OK) {
                // Depois que o usuário autorizou, faz login no Google Play Services
                if (!mGoogleApiClient.isConnecting() && !mGoogleApiClient.isConnected()) {
                    mGoogleApiClient.connect();
                }
            }
        }
    }

    private void toast(String s) {
        Log.d("livroandroid", "> " + s);
        Toast.makeText(getBaseContext(), s, Toast.LENGTH_SHORT).show();
    }
}

```

Capítulo 35 ■ Sensores e Google Fit

Esse código está um pouco simplificado para facilitar a leitura. No projeto de exemplo do livro você encontrará uma variável `authInProgress` no código, a qual controla se a janela de autorização está aberta, para não abrir a janela duas vezes.

No arquivo de layout teremos apenas um `TextView` para mostrar a quantidade de passos retornada pelo Google Fit.

 `/res/layout/activity_google_fit_pedometro.xml`

```
<LinearLayout ... android:orientation="vertical">
    <TextView android:text="@string/qtde_passos"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
</LinearLayout>
```

 `/res/values/strings.xml`

```
<resources>
    . . .
    <string name="qtde_passos">Quantidade de Passos</string>
</resources>
```

Para o código funcionar, é necessário habilitar a Fitness API na página de console do Google Developers (<https://console.developers.google.com>). Faça isso no menu **APIs & auth** > **APIs**. Lembre-se de que já fizemos isso nos capítulos 22 e 28, sobre mapas e GCM push. A figura 35.11 mostra estas três APIs habilitadas na minha conta.

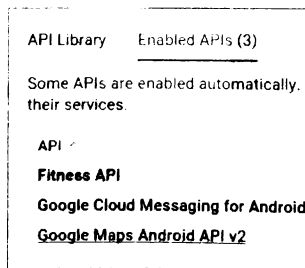


Figura 35.11 – Console do Google.

O próximo passo é gerar a chave de autenticação do Google Fit no menu **APIs & auth** > **Credentials**. Essa chave será do tipo **OAuth**, pois vamos precisar que o usuário entre com sua conta do Google no aplicativo para permitir o acesso. A autenticação do tipo **OAuth** permite ao aplicativo ler os dados do usuário, mas garante que suas

informações confidenciais fiquem privadas. Mas não se preocupe com isso, pois o Google controla tudo. Para gerar a chave, entre no menu **APIs & auth > Credentials** e clique no botão **Create New Client ID** na seção do **OAuth**.

Na janela que vai abrir (Figura 35.12), selecione o item **Installed application** e o tipo de aplicação Android. Digite o nome do pacote do aplicativo e o SHA1 Fingerprint do certificado. Lembre-se de que você aprendeu a obter o SHA1 Fingerprint no capítulo 22, sobre mapas. Feito isso, clique no botão **Create Client ID**. O legal dessa chave é que ela fica associada à sua conta do Google, porém não é preciso fazer nada no aplicativo. Basta criar essa configuração no servidor do Google, e está tudo pronto.

Create Client ID

Application type

- ☐ Web application
Accessed by web browsers over a network
- ☐ Service account
Calls Google APIs on behalf of your application instead of an end user. [Learn more](#)
- ☒ Installed application
Runs on a desktop computer or handheld device (like Android or iPhone)

Installed application type

- ☒ Android [Learn more](#)
- ☐ Chrome Application [Learn more](#)
- ☐ iOS [Learn more](#)
- ☐ PlayStation 4
- ☐ Other

API requests are sent directly to Google from your clients' Android devices. Google verifies that each request originates from an Android application that matches the package name and SHA1 signing certificate fingerprint name listed below.

Package name

br.com.livroandroid.sensores

Signing certificate fingerprint (SHA1)

C1 66 56 93 DF DE 0B B9 DC ED 76 D7 65 B7 10 DC 1F 3F 0D 41

Deep linking

- ☒ Enabled
- ☐ Disabled

Create Client ID Cancel

Figura 35.12 – Criando a chave OAuth.

Uma vez que a configuração e a chave **OAuth** foram criadas, vamos executar o aplicativo. Na primeira vez que você acessá-lo, será solicitado que o usuário faça o login com alguma conta do Google, conforme a figura 35.13. Isso aconteceu justamente porque foi criada uma chave de autenticação do tipo **OAuth**.

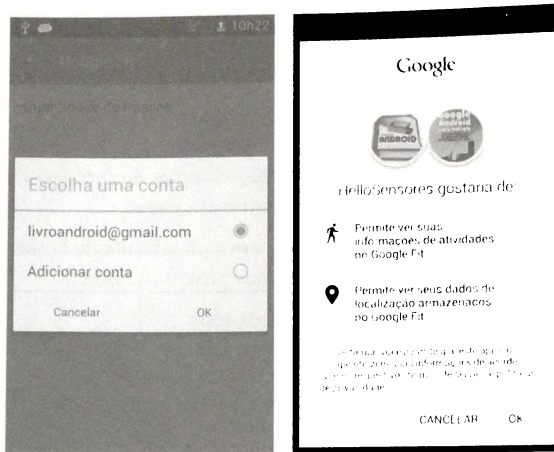


Figura 35.13 – Autenticando o usuário na aplicação.

Depois da autenticação, o método `onConnected(bundle)` será chamado e neste momento podemos iniciar a API do Google Fit. Se tudo correr bem, a API será iniciada e logo você receberá dados do sensor de contador de passos. Portanto, para testar, dê uma caminhada com o celular em mãos que a quantidade de passos será mostrada na tela, conforme a figura 35.14.

Repare que no código utilizamos a constante `DataType.TYPE_STEP_COUNT_DELTA` do Google Fit para receber os dados do sensor. Essa constante retorna a quantidade de passos feita desde a última leitura, portanto o aplicativo deve armazenar essa quantidade e fazer a soma de passos para mostrar a quantidade total. Mas lembre-se de que, se for necessário, podemos utilizar a **HistoryApi** para consultar os dados salvos na nuvem do Google.

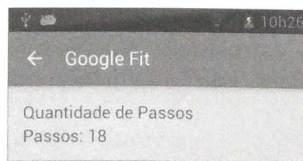


Figura 35.14 – Pedômetro em ação.

35.9 Links úteis

Neste capítulo estudamos como utilizar os sensores disponíveis na plataforma do Android.

Com a utilização de sensores é possível implementar funcionalidades bem interessantes nos aplicativos. Vimos até exemplos simples de como mover uma view com o acelerômetro, algo que é muito comum em jogos.

Também estudamos o Google Fit, que é a nova plataforma de Fitness do Google e talvez uma das áreas que mais crescerá nos próximos anos.

Seguem alguns links para continuar seus estudos.

- **API Guides – Sensors Overview**

http://developer.android.com/guide/topics/sensors/sensors_overview.html

- **Google Developers – Google Fit**

<https://developers.google.com/fit/>

- **Google Play Services – Fitness**

<https://developer.android.com/reference/com/google/android/gms/fitness/Fitness.html>



CAPÍTULO 36

Bluetooth

Bluetooth é muito utilizado como forma de comunicação entre dispositivos, mas seu uso é um tanto quanto trabalhoso.

Neste capítulo, vamos criar um aplicativo de chat capaz de enviar e receber mensagens, explicando todos os conceitos para facilitar o aprendizado.

36.1 Verificando se o dispositivo suporta Bluetooth

A principal classe da API de Bluetooth é a `BluetoothAdapter`, a qual representa o adaptador do Bluetooth disponível no dispositivo.

Para obter o `BluetoothAdapter`, utiliza-se o método `getDefaultAdapter()`. Embora a maioria dos dispositivos com Android tenha Bluetooth, é recomendado testar para verificar se o adaptador do hardware de Bluetooth existe no dispositivo, conforme demonstrado neste código:

```
// Bluetooth adapter
BluetoothAdapter btAdapter = BluetoothAdapter.getDefaultAdapter();
if(btAdapter != null) { // Existe Bluetooth }
else { // Não tem Bluetooth }
```

Depois de obter o objeto `BluetoothAdapter`, podemos executar diversas ações, como:

- fazer uma busca para encontrar dispositivos com o Bluetooth ativado;
- obter a lista de dispositivos pareados;
- conectar a outro dispositivo Bluetooth para enviar mensagens.

Nota: para testar os exemplos deste capítulo, recomendo ter dois dispositivos Android, pois vamos enviar mensagens de um para outro.

36.2 Ativando o Bluetooth por programação

Com o objeto `BluetoothAdapter` em mãos, o próximo passo é verificar se o Bluetooth está ativado no dispositivo. Isso é feito com o método `isEnabled()`, que retorna `true` caso o Bluetooth esteja ligado. Caso não esteja, podemos solicitar ao usuário para ativar as configurações do Bluetooth.

```
if (btAdapter.isEnabled()) { // Bluetooth está ligado }
else { // precisa ligar o Bluetooth }
```

Felizmente, é possível ativar as configurações do Bluetooth via programação. Para isso, precisamos declarar as permissões `BLUETOOTH` e `BLUETOOTH_ADMIN`:

AndroidManifest.xml

```
<manifest . . . >
    <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
    <uses-permission android:name="android.permission.BLUETOOTH" />
    <application . . . />
</manifest>
```

Com essas permissões, podemos utilizar as classes de Bluetooth e também disparar uma `Intent` que vai abrir um diálogo para o usuário ativar o Bluetooth:

```
Intent enableIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
startActivityForResult(enableIntent, 0);
```

Para demonstrar esse funcionamento, crie uma simples `activity` chamada `BluetoothCheckActivity` conforme demonstrado a seguir:

BluetoothCheckActivity.java

```
public class BluetoothCheckActivity extends AppCompatActivity {
    protected static final String TAG = "livroandroid";
    protected BluetoothAdapter btAdapter;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // Bluetooth adapter
        btAdapter = BluetoothAdapter.getDefaultAdapter();
        if (btAdapter == null) {
            Toast.makeText(this, "Bluetooth não disponível neste dispositivo.",
                Toast.LENGTH_LONG).show();
            // Vamos fechar a activity neste caso
            finish();
            return;
        }
    }
}
```

```

1
if (btAdapter.isEnabled()) { // Bluetooth Ok.
    Toast.makeText(this, "Bluetooth está ligado!", Toast.LENGTH_LONG).show();
} else { // De não está ligado, pede para o usuário ligar
    // Request permission and in return android.permission.BLUETOOTH_ADMIN" />
    Intent enableIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
    startActivityForResult(enableIntent, 0);
}
}
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (btAdapter.isEnabled()) {
        Toast.makeText(this, "Bluetooth foi ligado!", Toast.LENGTH_LONG).show();
    }
}

```

Agora vamos testar esse exemplo. Se você deixar o Bluetooth ligado, a activity vai mostrar uma toast informando que tudo está ok. Caso contrário, uma intent será disparada solicitando para o usuário ligar o Bluetooth. Caso ele responda **Sim**, o Bluetooth será ligado. A activity vai receber o retorno no método `onActivityResult()`, no qual podemos validar se de fato o Bluetooth foi ligado.

A figura 36.1 mostra a activity solicitando que o usuário ligue o Bluetooth.

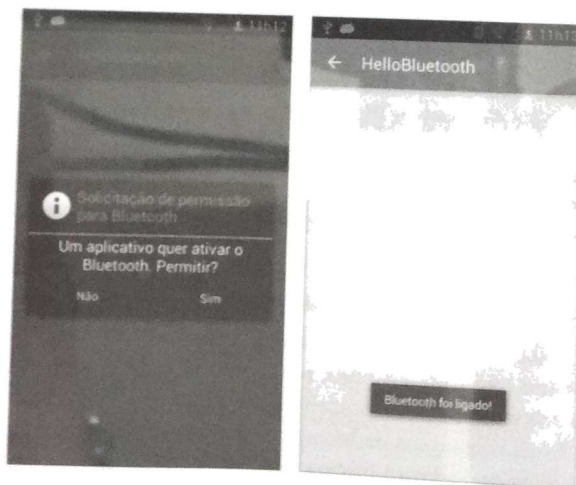


Figura 36.1 – Ativando o Bluetooth.

36.3 Listando os dispositivos pareados

Uma funcionalidade bem comum na comunicação por Bluetooth é a busca de dispositivos já pareados. Se você não está acostumado com Bluetooth, o pareamento é necessário antes de trocar informações entre um dispositivo e outro, sendo feito geralmente na primeira vez que esses dispositivos estabelecem uma conexão, informando um tipo de senha que ambos precisam aceitar. Depois que os dispositivos estão pareados, eles podem trocar informações.

Para recuperar a lista de dispositivos pareados, utilize o método `getBondedDevices()`:

```
Set<BluetoothDevice> pareados = btAdapter.getBondedDevices();
for (BluetoothDevice device : lista) {
    String nome = device.getName();
    String address = device.getAddress();
    // Fazer algo com essa informação...
}
```

Note que a lista é do tipo `BluetoothDevice`, objeto que representa um dispositivo. A classe `BluetoothDevice` não contém muitos métodos, os mais utilizados são o `getName()`, que retorna o nome do dispositivo, e o `getAddress()`, que retorna o endereço do hardware do Bluetooth. Esse endereço é único e por meio dele podemos nos conectar ao dispositivo.

A classe `ListaPareadosActivity` do projeto de exemplo deste capítulo mostra como exibir a lista de dispositivos pareados em um `ListView`. Mas como isso é simples demais, vamos pular para o próximo exemplo.

36.4 Buscar novos dispositivos Bluetooth

Para brincar com Bluetooth, precisamos buscar os dispositivos que estão com o Bluetooth ativado. Depois de encontrar algum dispositivo, é preciso fazer o pareamento para somente depois iniciar a conexão e troca de informações.

Para iniciar a busca, é utilizado o método `startDiscovery()` da classe `BluetoothAdapter` e para cancelar é utilizado o método `cancelDiscovery()`.

Os resultados da busca iniciadas pelo método `startDiscovery()` não são entregues por meio de um tradicional listener, e sim por meio de um `BroadcastReceiver`, o qual precisamos registrar dinamicamente na activity para receber broadcasts das mensagens. Essas mensagens de broadcast são disparadas pelo Android com o intuito de indicar que a busca iniciou (`DISCOVERY_STARTED`), que um dispositivo foi

Capítulo 36 ■ Bluetooth

encontrado (`ACTION_FOUND`) e que a busca terminou (`ACTION_DISCOVERY_FINISHED`). As mensagens com as ações `DISCOVERY_STARTED` e `ACTION_DISCOVERY_FINISHED` são chamadas apenas uma vez, para indicar o início e fim da busca. Mas a mensagem com ação `ACTION_FOUND` pode ser chamada várias vezes para cada dispositivo encontrado.

Agora, criaremos uma *activity* de exemplo que vai buscar os dispositivos Bluetooth e mostrá-los numa lista:



ListaDevicesActivity.java

```
...
public class ListaDevicesActivity extends BluetoothCheckActivity
    implements AdapterView.OnItemClickListener {
    private ProgressDialog dialog;
    protected List<BluetoothDevice> lista;
    private ListView listView;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_lista_devices);
        listView = (ListView) findViewById(R.id.listView);
        // Inicia a lista com os devices pareados
        lista = new ArrayList<BluetoothDevice>(btfAdapter.getBondedDevices());
        // Registra o receiver para receber as mensagens de dispositivos pareados
        this.registerReceiver(mReceiver, new IntentFilter(BluetoothDevice.ACTION_FOUND));
        // Registra o receiver para receber a mensagem do final da busca.
        this.registerReceiver(mReceiver, new
            IntentFilter(BluetoothAdapter.ACTION_DISCOVERY_FINISHED));
    }
    @Override
    protected void onResume() {
        super.onResume();    // Garante que não existe outra busca sendo realizada
        if (btfAdapter.isDiscovering()) { btfAdapter.cancelDiscovery(); }
        // Dispara a busca
        btfAdapter.startDiscovery();
        dialog = ProgressDialog.show(this, "Exemplo", "Buscando dispositivos Bluetooth...",
            false, true);
    }
    // Receiver para receber os broadcasts do Bluetooth
    private final BroadcastReceiver mReceiver = new BroadcastReceiver() {
        // Quantidade de dispositivos encontrados
        private int count;
        @Override
```



```

public void onReceive(Context context, Intent intent) {
    String action = intent.getAction();
    // Se um device foi encontrado
    if (BluetoothDevice.ACTION_FOUND.equals(action)) {
        // Recupera o device da intent
        BluetoothDevice device =
            intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
        // Apenas insere na lista os devices que ainda não estão pareados
        if (device.getBondState() != BluetoothDevice.BOND_BONDED) {
            lista.add(device);
            Toast.makeText(context, "Encontrou: " + device.getName()+";" +
                device.getAddress(), Toast.LENGTH_SHORT).show();
            count++;
        }
    } else if (BluetoothAdapter.ACTION_DISCOVERY_STARTED.equals(action)) {
        // Iniciou a busca
        count = 0;
        Toast.makeText(context, "Busca iniciada.", Toast.LENGTH_SHORT).show();
    } else if (BluetoothAdapter.ACTION_DISCOVERY_FINISHED.equals(action)) {
        // Terminou a busca
        Toast.makeText(context, "Busca finalizada. " + count +
            " devices encontrados", Toast.LENGTH_LONG).show();
        dialog.dismiss();
        // Atualiza o listview. Agora vai ter todos os devices pareados,
        // mais os novos que foram encontrados
        updateLista();
    }
}

};
@Override
protected void onDestroy() {
    super.onDestroy();
    // Garante que a busca é cancelada ao sair
    if (btAdapter != null) {
        btAdapter.cancelDiscovery();
    }
    // Cancela o registro do receiver
    this.unregisterReceiver(mReceiver);
}

private void updateLista() {
    // Cria o array com o nome de cada device
    List<String> nomes = new ArrayList<String>();

```

```

        for (BluetoothDevice device : lista) {
            // Neste exemplo, esta variável boolean sempre será true, pois esta lista é
            // somente dos pareados
            boolean pareado = device.getBondState() == BluetoothDevice.BOND_BONDED;
            nomes.add(device.getName() + " - " + device.getAddress() +
                (pareado ? " *pareado" : ""));
        }
        // Cria o adapter para popular o ListView
        int layout = android.R.layout.simple_list_item_1;
        ArrayAdapter<String> adapter = new ArrayAdapter<String>(this, layout, nomes);
        listView.setAdapter(adapter);
        listView.setOnItemClickListener(this);
    }
    @Override
    public void onItemClick(AdapterView<?> adapterView, View view, int idx, long id) {
        // Recupera o device selecionado
        BluetoothDevice device = lista.get(idx);
        String msg = device.getName() + " - " + device.getAddress();
        Toast.makeText(this, msg, Toast.LENGTH_SHORT).show();
    }
}

```

O arquivo de layout dessa activity contém apenas um ListView para mostrar a lista de dispositivos já pareados e aqueles que serão encontrados pela busca.

 /res/layout/activity_lista_devices.xml

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout . . android:orientation="vertical" >
    <ListView android:id="@+id/listView" android:layout_weight="1"
        android:layout_width="match_parent" android:layout_height="0dp" />
</LinearLayout>

```

A classe `ListaDevicesActivity` é filha da classe `BluetoothCheckActivity` que criamos anteriormente. Assim, herdamos o comportamento de verificar se o Bluetooth está ligado. Justamente por isso, o trecho de código que busca os dispositivos foi deixado no método `onResume()` do ciclo de vida, pois pode ser que o aplicativo saia da activity para solicitar ao usuário que ative o Bluetooth e, quando voltar, o método `onResume()` será utilizado para preencher a lista.

O código da activity inicia a lista com os dispositivos já pareados e depois dispara a busca com o método `startDiscovery()`. Quando um dispositivo é encontrado, a mensagem de broadcast com a ação `ACTION_FOUND` é enviada. Nesse momento, o

receiver que registramos no código intercepta a mensagem e recupera o objeto `BluetoothDevice` que representa o dispositivo encontrado.

A figura 36.2 mostra o resultado desse exemplo. Eu executei a activity em um Nexus S. Esse dispositivo já está pareado com um Nexus 5, portanto é o primeiro dispositivo da lista. Lembrando que os dispositivos pareados podem ser obtidos com o método `getBondedDevices()` e a lista foi inicializada com eles:

```
lista = new ArrayList<BluetoothDevice>(btfAdapter.getBondedDevices());
```

Durante a busca, foi encontrado um Nexus 6 que também foi adicionado na lista.

Quando a busca é finalizada, a mensagem com ação `ACTION_DISCOVERY_FINISHED` é enviada e interceptada pelo receiver. Nesse momento, foi mostrado um toast com a quantidade de dispositivos encontrados, que no meu caso foram três.

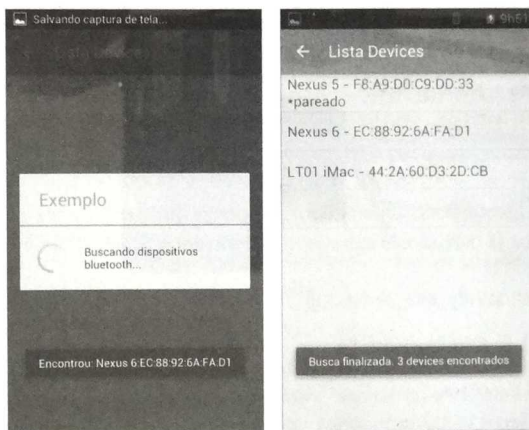


Figura 36.2 – Busca finalizada com dispositivos encontrados.

36.5 Deixando o Bluetooth visível para ser encontrado

No exemplo anterior fizemos a busca pelos dispositivos Bluetooth e adicionamos todos os resultados na lista. Porém, é importante você entender que somente dispositivos visíveis podem ser encontrados.

Para o dispositivo ser encontrado na busca, não basta o Bluetooth estar ligado. É preciso que ele esteja visível. Atualmente, no Android 4.0 ou superior, ao abrir a tela de configurações automaticamente, o dispositivo ficará visível. Mas caso seja

Capítulo 36 ■ Bluetooth

preciso deixar um dispositivo visível por programação, podemos disparar uma intent, a qual recebe como parâmetro o tempo em segundos que o dispositivo ficará visível para a busca:

```
// Deixa o dispositivo visível para ser encontrado na busca
Intent discoverableIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);
discoverableIntent.putExtra(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 300);
startActivity(discoverableIntent);
```

Essa intent solicita ao usuário que permita que seu dispositivo seja encontrado por outros durante o tempo em segundos informado. Ao dispará-la, o Android vai mostrar um alerta na janela conforme a figura 36.3. Se o usuário aceitar a solicitação, esse dispositivo poderá ser encontrado na busca.

Dica: caso você tenha executado o exemplo anterior e seu dispositivo não foi encontrado na busca, deixe-o visível e repita a busca.

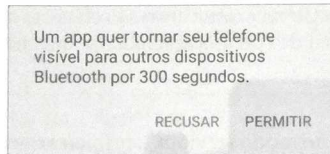


Figura 36.3 – Deixando o Bluetooth visível.

36.6 Criando um BluetoothDevice pelo endereço

A classe `BluetoothDevice` permite lermos o nome e o endereço Bluetooth de um dispositivo. Se o endereço Bluetooth de um dispositivo for previamente conhecido, podemos obter rapidamente o objeto `BluetoothDevice` com o método `getRemoteDevice(address)` sem a necessidade de uma busca.

O código-fonte a seguir mostra como obter um `BluetoothDevice` a partir de um endereço conhecido, que neste caso é o 40:FC:89:6D:0A:81.

```
BluetoothDevice device = btAdapter.getRemoteDevice("40:FC:89:6D:0A:81");
if(device != null) {
    String nome = device.getName();
    String endereco = device.getAddress();
}
```

Mas, na prática, geralmente os dispositivos são encontrados pela busca com o método `startDiscovery()` ou com o método `getBondedDevices()`, que retorna os dispositivos que já estão pareados. O importante é obter o objeto `BluetoothDevice` de alguma forma, pois por meio dele podemos iniciar uma conexão com esse dispositivo.

36.7 Chat em Bluetooth

Já aprendemos a usar o Bluetooth e buscar dispositivos. O próximo passo é criar uma conexão entre dois dispositivos para enviar e receber mensagens.

No Bluetooth uma conexão é realizada no modelo cliente-servidor. Um dispositivo inicia o modo servidor e fica aguardando conexões. Feito isso, outro dispositivo (cliente) conecta-se a esse servidor. Depois que a conexão é estabelecida, ambos os dispositivos podem abrir uma `InputStream` para ler mensagens e uma `OutputStream` para enviar mensagens.

Nota: as interfaces `InputStream` e `OutputStream` são clássicas do Java e permitem ler e escrever em um canal de comunicação aberto, que, neste caso, é a conexão Bluetooth.

Para iniciar o Bluetooth no modo servidor, a primeira tarefa é criar um objeto `UUID`. O importante é utilizar este mesmo `UUID` no aplicativo cliente para fazer a conexão.

```
private static final UUID uuid = UUID.fromString("fa87c0d0-afac-11de-8a39-0800200c9a66");
```

Nota: o `UUID` (Universally Unique Identifier) é um formato-padrão de texto com 128 bits e pode ser criado com qualquer gerador disponível na internet. Neste exemplo, o `UUID` vai identificar a conexão Bluetooth.

Com esse `UUID`, é possível chamar o método `listenUsingRfcommWithServiceRecord(string, uuid)` da classe `BluetoothAdapter` conforme demonstrado a seguir:

```
// Abre o socket servidor (quem for conectar precisa utilizar o mesmo UUID)
BluetoothServerSocket serverSocket = btAdapter.listenUsingRfcommWithServiceRecord(
    "LivroAndroid", uuid);
```

O retorno é um objeto do tipo `BluetoothServerSocket` que representa o servidor do socket criado. Feito isso, para definitivamente iniciar o servidor e aguardar as mensagens, o método `accept()` precisa ser chamado.

Capítulo 36 ■ Bluetooth

```
// Fica aguardando alguém conectar (esta chamada é bloqueante)
BluetoothSocket socket = serverSocket.accept();
```

O método `accept()` é bloqueante, isto é, a próxima linha será executada somente depois que algum cliente se conectar. Quando um dispositivo se conecta ao servidor, o objeto `BluetoothSocket` é retornado do método `accept()` e pode ser utilizado para obter a `InputStream` e a `OutputStream`.

Nota: se você já trabalhou com sockets em Java não terá nenhuma dificuldade para prosseguir a partir daqui. Basicamente vamos criar um servidor de socket com a classe `BluetoothServerSocket`, e quando um cliente se conectar é obtido o socket que é a classe `BluetoothSocket`. De resto, toda a comunicação é feita pelas interfaces `InputStream` e `OutputStream`, como de costume em aplicações Java.

Para encapsular a lógica do chat, vamos criar a classe `ChatController`, que vai receber o socket `BluetoothSocket` no construtor, assim como um listener para delegar os eventos quando uma mensagem for recebida.



ChatController.java

```
public class ChatController {
    private static final String TAG = "chat";
    private BluetoothSocket socket;
    private InputStream in;
    private OutputStream out;
    private ChatListener listener;
    private boolean running;
    public interface ChatListener {
        public void onMessageReceived(String msg);
    }
    public ChatController(BluetoothSocket socket, ChatListener listener) throws IOException {
        this.socket = socket;
        this.in = socket.getInputStream();
        this.out = socket.getOutputStream();
        this.listener = listener;
        this.running = true;
    }
    // Inicia a leitura da InputStream
    public void start() {
        new Thread(){
            @Override
```

```

    public void run() {
        running = true;
        // Faz a leitura
        byte[] bytes = new byte[1024];
        int length;
        // Fica em loop para receber as mensagens
        while (running) {
            try {
                Log.d(TAG, "Aguardando mensagem");
                // Lê a mensagem (fica bloqueado até receber)
                length = in.read(bytes);
                String msg = new String(bytes, 0, length);
                Log.d(TAG, "Mensagem: " + msg);
                // Recebeu a mensagem (informa o listener)
                listener.onMessageReceived(msg);
            } catch (Exception e) {
                running = false;
                Log.e(TAG, "Error: " + e.getMessage(), e);
            }
        }
    }
    }.start();
}

public void sendMessage(String msg) throws IOException {
    if (out != null) { out.write(msg.getBytes()); }
}

public void stop() {
    running = false;
    try {
        if (socket != null) { socket.close(); }
        if (in != null) { in.close(); }
        if (out != null) { out.close(); }
    } catch (IOException e) { }
}
}

```

Essa classe obtém a `InputStream` e a `OutputStream` do `BluetoothSocket` e inicia uma `thread` para ficar continuamente lendo as mensagens da `InputStream`. Sempre que uma mensagem é lida, o evento é entregue por meio da interface `ChatListener` com o método `onMessageReceived(msg)`. O método `sendMessage(msg)` dessa classe pode ser utilizado para enviar uma mensagem ao `socket` aberto, que na prática vai escrever os dados na `OutputStream`.

Agora vamos implementar a activity cliente, a qual vai receber um objeto do tipo `BluetoothDevice` por parâmetro da intent e vai se conectar ao dispositivo.

`BluetoothChatClientActivity.java`

```
public class BluetoothChatClientActivity extends BluetoothCheckActivity
    implements ChatController.ChatListener {
    protected static final String TAG = "livroandroid";
    // Precisa utilizar o mesmo UUID que o servidor utilizou para abrir o socket servidor
    protected static final UUID uuid = UUID.fromString("fa87c0d0-afac-11de-8a39-0800200c9a66");
    protected BluetoothDevice device;
    protected TextView tMsg, tMsgRecebidas;
    protected ChatController chat;
    @Override
    public void onCreate(Bundle icicle) {
        super.onCreate(icicle);
        setContentView(R.layout.activity_bluetooth_chat);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        tMsg = (TextView) findViewById(R.id.tMsg);
        tMsgRecebidas = (TextView) findViewById(R.id.tMsgRecebidas);
        // Device selecionado na lista
        device = getIntent().getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
        try {
            // Faz a conexão se abriu no modo chat cliente
            if(device != null) {
                getSupportActionBar().setTitle("Conectado: " + device.getName());
                // Faz a conexão utilizando o mesmo UUID que o servidor utilizou
                BluetoothSocket socket = device.createRfcommSocketToServiceRecord(uuid);
                socket.connect();
                // Inicia o controlador chat
                chat = new ChatController(socket, this);
                chat.start();
                findViewById(R.id.btEnviarMsg).setEnabled(true);
            }
        } catch (IOException e) {
            error("Erro ao conectar: " + e.getMessage(), e);
        }
    }
    public void onClickEnviarMsg(View view) {
        String msg = tMsg.getText().toString();
        try {
            chat.sendMessage(msg);
            tMsg.setText("");
        }
```



```

        // Mostra o texto enviado na área do chat
        String s = tMsgRecebidas.getText().toString();
        tMsgRecebidas.setText(s + "\n>> " + msg);
    } catch (IOException e) {
        error("Erro ao escrever: " + e.getMessage(), e);
    }
}

private void error(final String msg, final IOException e) {
    Log.e(TAG, "Erro no client: " + e.getMessage(), e);
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            Toast.makeText(getBaseContext(), msg, Toast.LENGTH_SHORT).show();
        }
    });
}

@Override
public void onMessageReceived(final String msg) {
    Log.d(TAG, "onMessageReceived (recebeu uma mensagem): " + msg);
    // É chamado numa thread, portanto use o runOnUiThread
    runOnUiThread(new Runnable() {
        @Override
        public void run() {
            String s = tMsgRecebidas.getText().toString();
            tMsgRecebidas.setText(s + "\n<< " + msg);
        }
    });
}

@Override
protected void onDestroy() {
    super.onDestroy();
    if(chat != null) {
        chat.stop();
    }
}
}

```

Essa classe recebe um objeto do tipo `BluetoothDevice` por parâmetro da intent e inicia uma conexão obtendo o `BluetoothSocket`. Depois de iniciar a conexão, a lógica do chat é delegada para a classe `ChatController`, a qual encapsula como ler e escrever mensagens na `InputStream` e `OutputStream`.

Capítulo 36 ■ Bluetooth

No layout desta activity vamos ter apenas um campo de texto para digitar uma mensagem e um botão para enviá-la. Na parte de baixo do layout, teremos um `TextView` que vai mostrar todas as mensagens trocadas no chat.

```

 /res/layout/activity_bluetooth_chat.xml

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout . . . android:orientation="vertical" >
    <LinearLayout android:layout_width="match_parent" android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <EditText android:id="@+id/tMsg"
            android:layout_width="0dp" android:layout_weight="1"
            android:layout_height="wrap_content"
            android:inputType="text" android:singleLine="true"/>
        <Button android:id="@+id/btEnviarMsg"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="@string/enviar_msg" android:enabled="false"
            android:onClick="onClickEnviarMsg" />
    </LinearLayout>
    <TextView android:id="@+id/tMsgRecebidas"
        android:layout_width="match_parent" android:layout_height="0dp"
        android:layout_weight="1" android:gravity="top"/>
</LinearLayout>

```

Como eu disse antes, a activity cliente vai receber um objeto do tipo `BluetoothDevice` por parâmetro pela intent. Logo, vamos alterar a classe `ListaDevicesActivity` de modo que, ao selecionarmos um dispositivo da lista, seja feita a navegação até a activity cliente do chat.

 `ListaDevicesActivity.java`

```

. . .
public class ListaDevicesActivity . . . {
    . . .
    public void onItemClick(AdapterView<?> adapterView, View view, int idx, long id) {
        // Recupera o device selecionado
        BluetoothDevice device = lista.get(idx);
        // Vai para a tela para enviar a mensagem
        Intent intent = new Intent(this, BluetoothChatClientActivity.class);
        intent.putExtra(BluetoothDevice.EXTRA_DEVICE, device);
        startActivity(intent);
    }
}

```

O código da parte cliente do chat está pronto, falta apenas o servidor. Sendo assim, crie a classe `BluetoothChatServerActivity`.



`BluetoothChatServerActivity.java`

```
public class BluetoothChatServerActivity extends BluetoothChatClientActivity
    implements ChatController.ChatListener {
    private static final UUID uuid = UUID.fromString("fa87c0d0-afac-11de-8a39-0800200c9a66");
    private boolean running;
    private BluetoothServerSocket serverSocket;
    @Override
    public void onCreate(Bundle icle) {
        super.onCreate(icle);
        // Deixa o servidor disponível para busca
        BluetoothUtil.makeVisible(this,300);
    }
    @Override
    protected void onResume() {
        super.onResume();
        // Inicia a thread do chat para não travar a UI
        new ChatThread().start();
    }
    class ChatThread extends Thread {
        @Override
        public void run() {
            try {
                // Abre o socket servidor (o cliente precisa utilizar o mesmo UUID)
                serverSocket =
                    btAdapter.listenUsingRfcommWithServiceRecord("LivroAndroid", uuid);
                Log.d(TAG, "Servidor aguardando conexão...");
                // Aguarda até alguém conectar (esta chamada é bloqueante)
                BluetoothSocket socket = serverSocket.accept();
                if (socket != null) {
                    // Mostra o device nome do device que conectou
                    final BluetoothDevice device = socket.getRemoteDevice();
                    runOnUiThread(new Runnable() {
                        @Override
                        public void run() {
                            getSupportActionBar().setTitle("Conectado: " + device.getName());
                            findViewById(R.id.btEnviarMsg).setEnabled(true);
                            Toast.makeText(getApplicationContext(),"Conectou: "
                                + device.getName(),Toast.LENGTH_SHORT).show();
                        }
                    });
                }
            }
        }
    }
}
```

```

        });
        // Alguém conectou
        chat = new ChatController(socket, BluetoothChatServerActivity.this);
        chat.start();
    }
} catch (IOException e) {
    Log.e(TAG, "Erro no servidor: " + e.getMessage(), e);
    running = false;
}
}
}
}
@Override
protected void onDestroy() {
    super.onDestroy();
    try {
        if (serverSocket != null) {
            serverSocket.close();
        }
    } catch (IOException e) {
    }
}
}
}

```

Essa classe inicia o servidor do chat abrindo um socket com a classe `BluetoothServerSocket`. Quando a parte cliente conecta-se ao socket, o método `serverSocket.accept()` retorna a conexão do socket no objeto `BluetoothSocket`. Logo depois, é iniciado o chat com a classe `ChatController`.

Pronto, terminamos o código do servidor. Inclusive ele utilizará o mesmo arquivo de layout do cliente, pois ambos precisam enviar e receber mensagens.

Naturalmente, o código que controla o processo de conexão é diferente entre servidor e cliente, mas ambos no final fazem a conexão e obtêm o socket `BluetoothSocket`. Com o socket em mãos, ambos, servidor e cliente, utilizam a classe `ChatController` para controlar a leitura e escrita das mensagens, e a partir disso o processo é o mesmo para ambos.

Note que a classe `BluetoothChatServerActivity` inclusive herda de `BluetoothChatClientActivity`, já que, como eu já disse, fora a parte de conexão, o restante do código entre as duas classes é o mesmo, pois ambos, servidor e cliente, podem enviar e receber mensagens.

Nota: o servidor do Bluetooth abre um socket servidor que aguarda conexões. Isso é feito pelo método `accept()`, que ficará bloqueado até algum cliente se conectar. Depois que a conexão é realizada, o objeto `BluetoothSocket` representa a conexão e, por meio dele, podemos obter a `InputStream` e a `OutputStream` para ler e escrever mensagens.

Agora vamos executar o chat em Bluetooth para brincarmos um pouco e depois recomendo que você estude bem o código. Naturalmente, para continuar, tenha em mãos dois dispositivos Android. Nos exemplos que vou mostrar o servidor foi aberto em um Nexus 5 e o cliente em um Nexus S.

No projeto de exemplo do livro temos a lista com os exemplos conforme a figura 36.4. Clique no exemplo **Iniciar servidor Chat**, o qual é exatamente a classe `BluetoothChatServerActivity` que acabamos de estudar. Feito isso, o servidor do chat vai abrir e aguardar uma conexão. Veja que é importante que o servidor esteja visível, caso o cliente precise fazer uma busca para encontrá-lo.

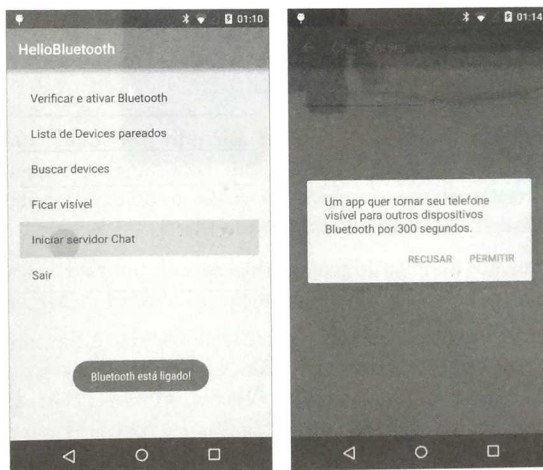


Figura 36.4 – Iniciado o servidor do chat.

Logo depois, pegue o outro celular e faça uma busca pelos dispositivos conforme a figura 36.5. No meu caso foi encontrado o celular Nexus 5, que é o servidor.

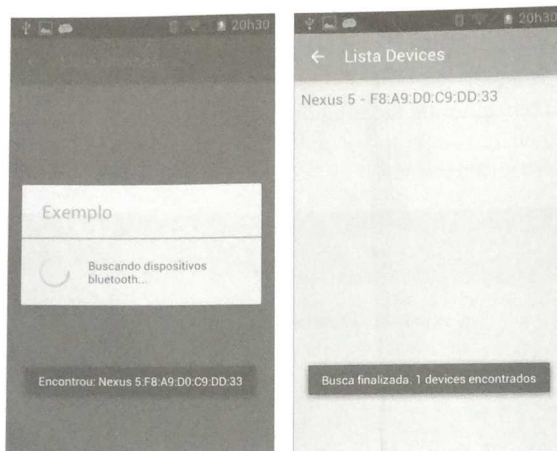


Figura 36.5 – Buscando os dispositivos.

Ao clicar na lista, vamos chamar a activity cliente do chat `BluetoothChatClientActivity`, que vai iniciar a conexão entre os dois dispositivos. Caso os dispositivos ainda não estejam pareados, é feito o pareamento em ambos, conforme a figura 36.6. Note que, caso os dispositivos já estejam pareados, não é necessário nem fazer a busca, pois podemos obter rapidamente os dispositivos já conhecidos.

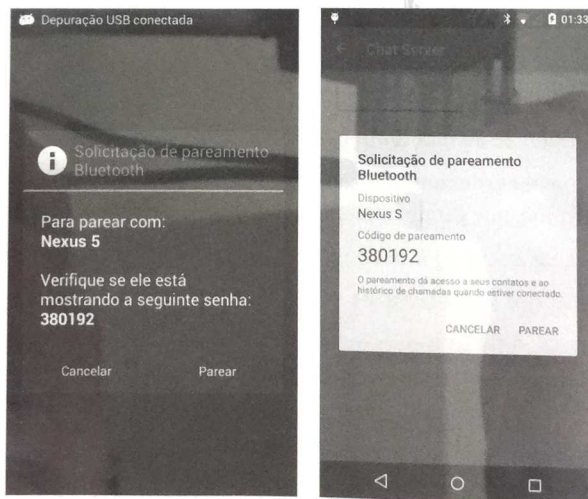


Figura 36.6 – Pareando os dispositivos.

Depois de parear os dispositivos, a conexão é realizada. Em ambos os dispositivos (cliente e servidor), o aplicativo mostra no título da action bar o nome do outro dispositivo. Se a conexão for bem-sucedida, o botão **Enviar Mensagem** é habilitado, então basta começar a digitar. A figura 36.7 mostra uma conversa entre os dois celulares. No chat, o símbolo ">>" indica uma mensagem enviada; e o símbolo "<<", uma mensagem recebida.

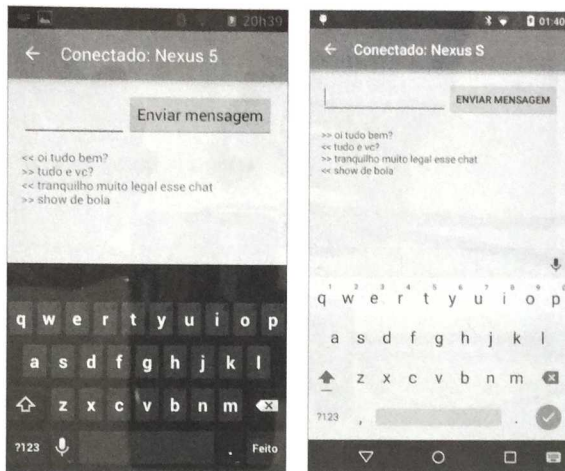


Figura 36.7 – Enviando mensagens no chat.

36.8 Conectando-se ao Bluetooth pela serial

Bluetooth é um meio de comunicação também muito utilizado para se comunicar com um Arduino, que geralmente utiliza a conexão serial.

Para se conectar à serial por Bluetooth, utilize o UUID padrão da serial:

```
00001101-0000-1000-8000-00805F9B34FB
```

Basicamente, você irá utilizar o método `createRfcommSocketToServiceRecord(UUID)` da classe `BluetoothDevice`, mas vai passar esse UUID padrão da serial:

```
UUID uuidSerial = UUID.fromString("00001101-0000-1000-8000-00805F9B34FB");
BluetoothSocket socket = device.createRfcommSocketToServiceRecord(uuidSerial);
socket.connect();
OutputStream out = socket.getOutputStream();
```

Capítulo 36 ■ Bluetooth

Caso esteja curioso de onde tirei esse UUID padrão da serial, isso está escrito na documentação (javadoc) da classe `BluetoothDevice`.

Depois de realizar essa conexão, você pode enviar os dados por Bluetooth, mas por de baixo dos panos a conexão é realizada como se fosse serial.

Essa abordagem está sendo muito utilizada na comunicação de dispositivos Android com Arduino que geralmente recebem os dados pela serial. Você deve apenas instalar uma placa receptora de Bluetooth no Arduino para realizar a conexão, mas isso é com você! Aqui estou dando apenas a dica caso algum dia você precise fazer isso.

36.9 Links úteis

Neste capítulo estudamos a comunicação por Bluetooth e inclusive desenvolvemos um simples chat.

Segue um link para continuar seus estudos.

- **API Guides – Bluetooth**

<http://developer.android.com/guide/topics/connectivity/bluetooth.html>



CAPÍTULO 37

Reconhecimento de voz

Reconhecimento de voz está sendo cada vez mais importante nos aplicativos para dispositivos móveis, principalmente para relógios (Android Wear), óculos (Google Glass) e carros (Android Auto), pois todas essas plataformas estão utilizando cada vez mais os comandos de voz.

Neste capítulo vamos aprender a fazer o Android falar e escutar.

37.1 Introdução

Neste capítulo, vamos aprender a fazer o Android falar e escutar e usufruir dos recursos de voz que são cada vez mais importantes para novas plataformas e dispositivos como relógios, óculos, carros etc.

Text-To-Speech (TTS) é a API responsável por converter texto em voz e fazer o Android falar, e Speech-To-Text (STT) é responsável por converter voz em texto e fazer o Android escutar.

O reconhecimento de voz (STT) é muito útil para interagir com os dispositivos quando não é possível tocá-los, como é o caso de enviar comandos para o celular quando estamos dirigindo.

Já a capacidade do Android de falar (TTS) é muito útil para auxiliar na navegação do aplicativo quando não podemos prestar atenção na tela. Novamente, é o caso de quando estamos dirigindo, pois o aplicativo do GPS pode falar para nos orientar sobre as informações da rota.

Em especial, o TTS é muito utilizado para criar aplicativos com acessibilidade para deficientes visuais. Eu recomendo assistir à palestra “Text-To-Speech & Eyes-Free Project” no YouTube, apresentada por dois engenheiros do Google durante o Google I/O 2009. O interessante da palestra é que um dos palestrantes era

deficiente visual e, dentre outras coisas, ele mostrou como esse tipo de tecnologia pode mudar a vida de milhares de pessoas pelo mundo.

37.2 Hello TTS – faça seu Android falar

Para fazer o Android falar, podemos utilizar a classe `TextToSpeech`, que está presente no Android desde a versão 1.6. A engine de voz está disponível para inglês, alemão, espanhol, francês e italiano, mas tudo pode variar de acordo com o fabricante. Infelizmente, não são todos os aparelhos com Android que suportam a voz em português ou, como dizemos tecnicamente, a localização (locale) `pt-BR`.

A API do TTS é simples e tudo gira ao redor da classe `TextToSpeech`, que recebe no construtor uma implementação da interface `TextToSpeech.OnInitListener`.

```
TextToSpeech tts = new TextToSpeech(this, this); // Aguarde o método onInit()
```

Logo após criar uma instância da classe `TextToSpeech`, é preciso aguardar a inicialização da engine de voz. Quando o método `onInit(status)` da interface de listener for chamado, significa que a engine foi inicializada com sucesso, e assim podemos usar o método `speak(texto, modo, params)` informando o texto para falar.

```
tts.speak("Olá Tudo bem?", TextToSpeech.QUEUE_FLUSH, null);
```

Para configurar o locale utilizado pela engine do TTS, utilize o método `setLocale(locale)` conforme demonstrado a seguir:

```
// Português do Brasil
Locale locale = new Locale("pt", "BR");
tts.setLanguage(locale);
// Inglês padrão
tts.setLanguage(Locale.ENGLISH);
```

O TTS também pode sintetizar a voz e salvar em um arquivo de áudio. Isso é feito com o método `synthesizeToFile(texto, params, arquivo)`.

```
File file = SDCardUtils.getPublicFile("arquivo-voz.wav", Environment.DIRECTORY_MUSIC);
HashMap<String, String> params = new HashMap<String, String>();
tts.synthesizeToFile("Olá, tudo bem?", params, file.getAbsolutePath());
```

Por último, quando a engine do TTS não for mais utilizada, você deve chamar o método `shutdown()` para liberar os recursos. Isso geralmente é feito no método `onDestroy()` da `activity`.

```
// Libera os recursos da engine do TTS
tts.shutdown();
```

Outro detalhe importante é que muitas vezes precisamos verificar se o pacote de dados de voz está instalado no aplicativo. Isso pode ser feito disparando uma intent:

```
// Verifica se o pacote de dados do TTS está instalado
Intent checkIntent = new Intent();
checkIntent.setAction(TextToSpeech.Engine.ACTION_CHECK_TTS_DATA);
startActivityForResult(checkIntent, ACTION_CHECK_DATA_CODE);
```

O resultado é entregue no método `onActivityResult(...)`. Se o código de retorno for `ACTION_CHECK_DATA_CODE`, isso indica que o pacote de dados de voz está instalado. Caso contrário, podemos instalá-lo também com uma intent, a qual vai abrir uma aplicação para o usuário fazer o download do pacote de voz para o idioma desejado.

```
// Solicita a instalação do pacote de dados de voz
Intent installIntent = new Intent();
installIntent.setAction(TextToSpeech.Engine.ACTION_INSTALL_TTS_DATA);
startActivity(installIntent);
```

Depois desta introdução, vamos criar um exemplo prático. Crie a activity `HelloTTSActivity` com o seguinte arquivo de layout. Se preferir, abra o projeto `HelloVoz` disponível nos exemplos do livro.

 `/res/layout/activity_hello_tts.xml`

```
<LinearLayout . . . android:orientation="vertical">
    <EditText android:id="@+id/tMsg"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="Olá, bom dia." />
    <Button android:onClick="onClickFalarTexto"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Falar o Texto" />
    <Button android:onClick="onClickSalvar"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Salvar Arquivo" />
    <Button android:onClick="onClickFalarArquivo"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Falar do Arquivo" />
</LinearLayout>
```

No layout temos um campo para digitar um texto. O botão **Falar o Texto** vai utilizar a engine do TTS para falar. O botão **Salvar Arquivo** vai utilizar a engine da mesma forma, mas em vez de falar, vai salvar a voz em um arquivo de áudio no SD card. E o botão **Falar do Arquivo** vai tocar o arquivo de áudio salvo com a classe `MediaPlayer`. O código-fonte da activity pode ser visualizado a seguir:



HelloTTSActivity.java

```

...
public class HelloTTSActivity extends BaseActivity implements TextToSpeech.OnInitListener {
    private static final String TAG = "livroandroid";
    private static final int ACTION_CHECK_DATA_CODE = 1;
    private TextView tMsg;
    private TextToSpeech tts;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_tts);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        tMsg = (TextView) findViewById(R.id.tMsg);
        tts = new TextToSpeech(this, this);
    }
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.menu_hello_tts, menu);
        return true;
    }
    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        int id = item.getItemId();
        if (id == R.id.action_pt_br) {
            Locale locale = new Locale("pt", "BR");
            tts.setLanguage(locale);
            return true;
        } else if (id == R.id.action_en_us) {
            Locale locale = Locale.ENGLISH;
            tts.setLanguage(locale);
            return true;
        } else if (id == R.id.action_check_data) {
            // Verifica se o pacote de dados do TTS está instalado
            Intent checkIntent = new Intent();
            checkIntent.setAction(TextToSpeech.Engine.ACTION_CHECK_TTS_DATA);
            startActivityForResult(checkIntent, ACTION_CHECK_DATA_CODE);
            return true;
        } else if (id == R.id.action_install_data) {
            // Instala o pacote de dados
            Intent installIntent = new Intent();
            installIntent.setAction(TextToSpeech.Engine.ACTION_INSTALL_TTS_DATA);
            startActivity(installIntent);
        }
    }
}

```

```

        return true;
    }
    return super.onOptionsItemSelected(item);
}
@Override
public void onInit(int status) {
    Log.d(TAG, "Engine TTS inicializada com sucesso: " + Locale.getDefault());
    // Deixa inglês por padrão
    tts.setLanguage(Locale.getDefault());
}
public void onClickFalarTexto(View view) {
    String s = tMsg.getText().toString();
    tts.speak(s, TextToSpeech.QUEUE_FLUSH, null);
    Log.d(TAG, "Speak: " + s);
}
public void onClickSalvar(View view) {
    String s = tMsg.getText().toString();
    // Sintetiza a voz para arquivo
    File file = SDCardUtils.getPublicFile("arquivo-voz.wav", Environment.DIRECTORY_MUSIC);
    HashMap<String, String> params = new HashMap<String, String>();
    tts.synthesizeToFile(s, params, file.getAbsolutePath());
    toast("Voz salva em arquivo: " + file);
}
public void onClickFalarArquivo(View view) {
    File file = SDCardUtils.getPublicFile("arquivo-voz.wav", Environment.DIRECTORY_MUSIC);
    if(file.exists()) {
        try {
            MediaPlayer mp = new MediaPlayer();
            mp.setDataSource(file.getAbsolutePath());
            mp.prepare();
            mp.start();
        } catch (Exception e) {
            Log.e(TAG, "Erro ao tocar: " + e.getMessage(), e);
        }
    }
}
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == ACTION_CHECK_DATA_CODE) {
        if (resultCode == TextToSpeech.Engine.CHECK_VOICE_DATA_PASS) {
            toast("Pacote de dados de voz OK!");
        } else {

```

```

        // Falta pacote, solicita instalação
        Intent installIntent = new Intent();
        installIntent.setAction(TextToSpeech.Engine.ACTION_INSTALL_TTS_DATA);
        startActivity(installIntent);
    }
}
@Override
protected void onDestroy() {
    super.onDestroy();
    // Libera os recursos da engine do TTS
    tts.shutdown();
}
}

```

Crie o seguinte arquivo de menu com as opções da action bar.

 /res/layout/menu_hello_tts.xml

```

<menu . . .>
    <item android:id="@+id/action_pt_br"
        android:title="@string/action_pt_br" app:showAsAction="never" />
    <item android:id="@+id/action_en_us"
        android:title="@string/action_en_us" app:showAsAction="never" />
    <item android:id="@+id/action_check_data"
        android:title="@string/action_check_data" app:showAsAction="never" />
    <item android:id="@+id/action_install_data"
        android:title="@string/action_install_data" app:showAsAction="never" />
</menu>

```

O método `synthesizeToFile(texto,params,arquivo)` precisa das seguintes permissões para funcionar; sendo assim, adicione-as no arquivo de manifesto.

 AndroidManifest.xml

```

<manifest . . . >
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.RECORD_AUDIO"/>
    <application . . . />
</manifest>

```

A figura 37.1 mostra a aplicação executando no emulador. Veja que no menu temos algumas ações, como trocar o idioma da engine de voz, assim como verificar e instalar o pacote de dados de voz.

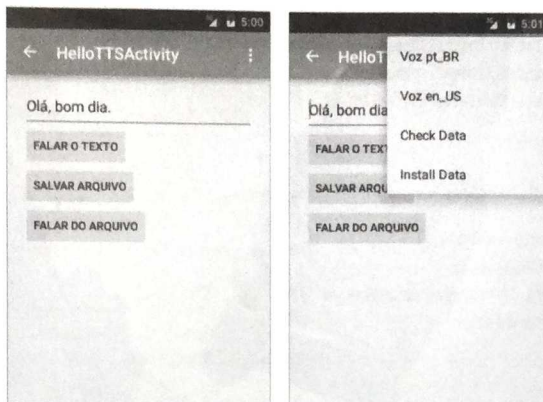


Figura 37.1 – Exemplo de TTS.

Executar a aplicação e testar os resultados é com você, pois não posso fazer o livro falar. Mas, antes de encerrarmos o assunto, vamos explicar alguns detalhes importantes. Ao criar a classe `TextToSpeech`, é preciso informar uma implementação da interface `OnInitListener`. Quando o método `onInit(status)` for chamado, significa que a engine de voz foi inicializada. O parâmetro `status` pode ser `TextToSpeech.SUCCESS` em caso de sucesso ou `TextToSpeech.ERROR` em caso de falha.

No caso de a inicialização ser bem-sucedida, podemos chamar o método `speak(texto,modo,params)`, passando a frase para a engine de voz falar.

Argumento	Descrição
String texto	Texto para falar.
int queueMode	Determina como o texto será processado pela fila de execução da engine. Aceita as constantes <code>TextToSpeech.QUEUE_FLUSH</code> e <code>TextToSpeech.QUEUE_ADD</code> . A constante <code>TextToSpeech.QUEUE_FLUSH</code> faz com que todos os textos que ainda não foram pronunciados sejam descartados da fila e o novo texto seja falado imediatamente. Já a constante <code>TextToSpeech.QUEUE_ADD</code> faz com que o novo texto seja adicionado à fila, para aguardar o término da pronúncia dos outros textos que estiverem na fila.
HashMap params	Parâmetros opcionais a serem passados para a engine, como <code>KEY_PARAM_STREAM</code> , <code>KEY_PARAM_UTTERANCE_ID</code> , <code>KEY_PARAM_VOLUME</code> , <code>KEY_PARAM_PAN</code> . Para mais detalhes, verificar a documentação oficial.

O método `speak(texto,modo,params)` foi descontinuado (deprecated) na API Level 21, pois foi substituído pelo método `speak(text, queueMode, params, utteranceId)`. Utilizamos o método antigo no código por questões de compatibilidade.

Capítulo 37 ■ Reconhecimento de voz

Argumento	Descrição
String texto	Idem ao anterior.
int queueMode	Idem ao anterior.
HashMap params	Idem ao anterior, mas agora os parâmetros são passados por um Bundle.
String utteranceId	Identificador único desta chamada.

Outro método que também utilizamos no código é o `synthesizeToFile(texto,params,arquivo)`, que converte o texto para voz e salva o áudio em um arquivo.

Argumento	Descrição
String texto	Texto para falar.
HashMap params	Parâmetros opcionais como no método <code>speak(...)</code> .
String fileName	Caminho do arquivo para salvar o áudio.

Esse método também foi descontinuado (deprecated) na API Level 21, sendo substituído pelo método `synthesizeToFile(texto,params,arquivo)`.

Argumento	Descrição
String texto	Texto para falar.
HashMap params	Parâmetros opcionais como no método <code>speak(...)</code> .
File file	Arquivo para salvar o áudio.
String utteranceId	Identificador único desta chamada.

Neste tópico aprendemos a fazer o Android falar. O assunto ainda é novo e pouco explorado. Com certeza, se você for desenvolver alguma aplicação real de voz, será necessário se aprofundar nos estudos, pois o que fizemos aqui foi um simples hello world para te mostrar o básico da API.

Nota: internamente, o TTS é um Service compartilhado entre todas as aplicações, sendo assim é recomendado liberar os recursos quando você não os utilizar mais. Faça isso chamando o método `shutdown()`.

37.3 Verificando o idioma e falando em português

Por padrão, a engine de voz não suporta o português, embora ao comercializarem o aparelho no Brasil alguns fabricantes adicionem o suporte ao português. Mas, como nada é garantido, vamos explicar o passo a passo básico para fazer seu Android falar em português.

Teoricamente, se quisermos falar em português, basta alterar o locale da engine para pt-BR:

```
TextToSpeech tts = . . . ;  
tts.setLanguage(new Locale("pt", "BR"));
```

Infelizmente, não é assim tão simples, pois o português não é suportado por padrão e tudo vai depender da customização que foi feita em seu aparelho.

Portanto, para ter certeza de que um idioma é suportado, chame o método `isLanguageAvailable(locale)`, que retorna uma das seguintes constantes:

```
int LANG_AVAILABLE = 0
```

Indica que o idioma é suportado, mas não exatamente daquele país. Por exemplo, pode ser que o português de Portugal seja suportado, mas não o do Brasil. Por isso, lembre-se de que um locale é geralmente especificado com o código do idioma e do país, como `new Locale("pt", "BR")`.

```
int LANG_COUNTRY_AVAILABLE = 1
```

Indica que o locale é suportado exatamente da forma que foi informado. Se essa constante for retornada, tudo está ok.

```
int LANG_COUNTRY_VAR_AVAILABLE = 2
```

Idem à constante `LANG_COUNTRY_AVAILABLE`. A diferença é que esta constante também leva em consideração as variantes da classe `Locale`, o que pode ser especificado no terceiro parâmetro do construtor, como `new Locale("pt", "BR", variantes)`.

```
int LANG_MISSING_DATA = -1
```

Indica que o idioma é suportado, mas falta fazer o download dos pacotes de voz. Caso seja necessário, podemos disparar uma intent para fazer esse download. Mas, geralmente, esse retorno acontece somente para os idiomas suportados de forma nativa pelo Android.

```
int LANG_NOT_SUPPORTED = -2
```

Infelizmente, o idioma especificado não é suportado.

Caso seu celular não tenha a engine de voz para português, é possível instalar alguma disponível no Google Play. A mais famosa é a **SVOX Classic**, que tem um sintetizador para português. Depois de instalar o aplicativo, você vai precisar baixar os arquivos de voz e configurar nas preferências do TTS a voz para português, mas você pode encontrar esse passo a passo na internet.

37.4 Reconhecimento de voz por intent

Até o momento, estamos fazendo o Android falar, mas agora vamos fazê-lo escutar. Desta vez, vamos converter a voz em texto, o que é conhecido como STT (Speech-To-Text).

Para a alegria dos desenvolvedores, o reconhecimento de voz pode ser feito por uma simples intent:

```
// Intent para fazer o reconhecimento de voz
Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
    RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
intent.putExtra(RecognizerIntent.EXTRA_CALLING_PACKAGE, getPackageName());
intent.putExtra(RecognizerIntent.EXTRA_PROMPT, "Fale algo");
intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE, "pt-BR");
intent.putExtra(RecognizerIntent.EXTRA_MAX_RESULTS, 10);
startActivityForResult(intent, 0);
```

O parâmetro `EXTRA_PROMPT` recebe o texto que será mostrado no alerta padrão do Android, o parâmetro `EXTRA_LANGUAGE` recebe o idioma para auxiliar no reconhecimento de voz e o parâmetro `EXTRA_MAX_RESULTS` recebe a quantidade máxima de resultados que devem ser entregues. O resultado da intent é entregue no método `onActivityResult(...)`, no qual podemos obter uma lista das possíveis palavras que foram encontradas:

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (resultCode == RESULT_OK) {
        // Recupera as possíveis palavras que foram pronunciadas
        ArrayList<String> words = data.getStringArrayListExtra(RecognizerIntent.EXTRA_RESULTS);
        listView.setAdapter(new ArrayAdapter<String>(this,
            android.R.layout.simple_list_item_1, words));
    }
}
```

Para demonstrar como implementar o reconhecimento de voz, vamos criar uma tela com um botão para disparar essa intent e um `ListView` para exibir a lista com os resultados. O arquivo de layout pode ser visualizado a seguir:

 /res/layout/activity_hello_recognizer_intent.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout ... android:orientation="vertical" >
    <Button android:id="@+id/btSpeak"
        android:layout_width="match_parent" android:layout_height="wrap_content"
        android:text="Clique aqui e fale algo" />
    <ListView android:id="@+id/list" android:layout_weight="1"
        android:layout_width="match_parent" android:layout_height="0dp" />
</LinearLayout>
```

Ao clicar no botão, será disparada uma intent para chamar o aplicativo nativo de reconhecimento de voz. O resultado será mostrado no `ListView`. Note que, como o reconhecimento de voz não é suportado em todos os aparelhos, o código está verificando se ele é suportado antes de disparar a intent.

 **HelloRecognizerIntentActivity.java**

```
...
public class HelloRecognizerIntentActivity extends AppCompatActivity {
    protected ListView listView;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_recognizer_intent);
        View btSpeak = findViewById(R.id.btSpeak);
        listView = (ListView) findViewById(R.id.list);
        // Verifica se o Android suporta a intent de reconhecimento de voz
        PackageManager pm = getPackageManager();
        List<ResolveInfo> activities = pm.queryIntentActivities(new
            Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH), 0);
        if (activities.size() != 0) {
            btSpeak.setOnClickListener(onClickSpeak());
            btSpeak.setEnabled(true);
        } else {
            Toast.makeText(this, "Reconhecimento de voz indisponível", Toast.LENGTH_SHORT).show();
        }
    }
    protected View.OnClickListener onClickSpeak() {
        return new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = getRecognizerIntent();
```

```

        startActivityResult(intent, 0);
    }
};
}
// Intent que dispara o reconhecimento de voz
protected Intent getRecognizerIntent() {
    Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
    intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
        RecognizerIntent.LANGUAGE_MODEL_FREE_FORM);
    intent.putExtra(RecognizerIntent.EXTRA_CALLING_PACKAGE, getPackageName());
    intent.putExtra(RecognizerIntent.EXTRA_PROMPT, "Fale algo");
    intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE, "pt-BR");
    intent.putExtra(RecognizerIntent.EXTRA_MAX_RESULTS, 10);
    return intent;
}
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (resultCode == RESULT_OK) {
        // Recupera as possíveis palavras que foram pronunciadas
        ArrayList<String> words = data.getStringArrayListExtra(RecognizerIntent.EXTRA_RESULTS);
        listView.setAdapter(new ArrayAdapter<String>(this,
            android.R.layout.simple_list_item_1, words));
    }
}
}

```

Nota: o reconhecimento de voz vai utilizar os servidores do Google para fazer a conversão de voz para texto, portanto certifique-se de que seu Android está conectado à internet.

Ao clicar no botão, a intent é disparada para iniciar o reconhecimento de voz, conforme demonstra a figura 37.2. Neste caso eu disse “*Olá, pessoal*”, e na lista pode-se ver que foi encontrado justamente esse texto. Agora é com você, fale algumas coisas e brinque com o seu Android.

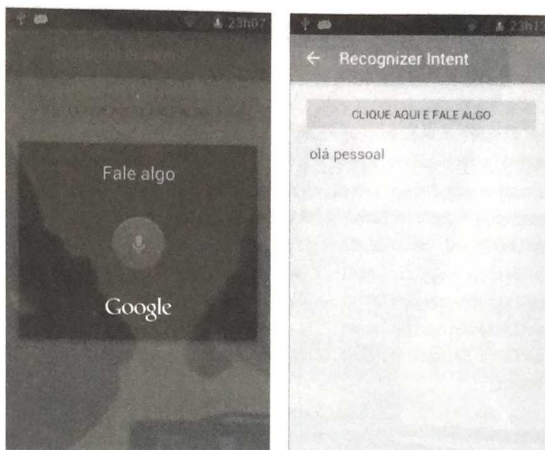


Figura 37.2 – Reconhecimento de voz.

37.5 Reconhecimento de voz por um listener

Sempre que possível, implementar uma funcionalidade com intents pode simplificar muito o aplicativo e economizar várias linhas de código. Mas utilizar uma intent implica mostrar a janela para o usuário falar, o que dependendo dos requisitos do projeto não é algo viável. Sendo assim, também é possível fazer o reconhecimento de voz pela API em segundo plano, sem utilizar intents.

A classe `SpeechRecognizer` é utilizada para fazer o reconhecimento de voz pela API. Basta configurá-la com um listener do tipo `RecognitionListener` e chamar o método `startListening(intent)`. Feito isso, a engine do reconhecimento de voz fica executando em segundo plano e lendo qualquer coisa que você falar.

```
SpeechRecognizer stt = SpeechRecognizer.createSpeechRecognizer(this);  
stt.setRecognitionListener(listener);  
Intent intent = . . . ;  
stt.startListening(intent);
```

O método `startListening(intent)` recebe como parâmetro a mesma intent que utilizamos no tópico anterior. Como a interface `RecognitionListener` contém vários métodos, vamos criar a classe `BaseRecognitionListener` que implementa essa interface com métodos vazios para facilitar o código final.



BaseRecognitionListener.java

```

. . .
public class BaseRecognitionListener implements RecognitionListener {
    // Indica que o usuário começou a falar
    public void onBeginningOfSpeech() { }
    // Buffer que vai sendo montado à medida que o usuário vai falando
    // Não é garantido que este método seja chamado
    public void onBufferReceived(byte[] buffer) { }
    // Indica que o usuário terminou de falar
    public void onEndOfSpeech() { }
    // Indica que ocorreu um erro no reconhecimento de voz
    public void onError(int error) { }
    // No momento este método não é utilizado
    public void onEvent(int eventType, Bundle params) { }
    // Chamado quando existe alguma leitura de voz parcial
    // Não é garantido que este método seja chamado
    public void onPartialResults(Bundle partialResults) { }
    // Chamado quando a aplicação está pronta para receber o comando de voz
    public void onReadyForSpeech(Bundle params) { }
    // Chamado para entregar os resultados para a aplicação
    public void onResults(Bundle results) { }
    // Indica que o nível de som do áudio mudou
    public void onRmsChanged(float rmsdB) { }
}

```

O método mais importante dessa interface é o `onResults(Bundle)`, o qual é chamado quando alguma palavra pronunciada é reconhecida. Neste exemplo, esse método é o único que vamos sobrescrever. Agora que temos essa classe que simplifica a interface `RecognitionListener`, podemos utilizá-la com a classe `SpeechRecognizer`:

```

SpeechRecognizer stt = SpeechRecognizer.createSpeechRecognizer(this);
stt.setRecognitionListener(new BaseRecognitionListener(){
    public void onResults(Bundle results) {
        // Recupera as possíveis palavras que foram pronunciadas
        ArrayList<String> words = results.getStringArrayList(
            SpeechRecognizer.RESULTS_RECOGNITION);
        listView.setAdapter(new ArrayAdapter<String>(getContext(),
            android.R.layout.simple_list_item_1, words));
    }
});

```

Como podemos verificar, o método `onResults(Bundle)` recebe o `Bundle` que contém a lista de palavras que foram pronunciadas. O código-fonte completo desse exemplo pode ser visualizado a seguir:



HelloSpeechRecognizerActivity.java

```
public class HelloSTT_Listener extends AppCompatActivity {
    // Reconhecedor de voz
    private SpeechRecognizer stt;
    private ListView listView;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        setContentView(R.layout.activity_hello_speech_recognizer);
        listView = (ListView) findViewById(R.id.list);
        // Cria o SpeechRecognizer e configura o listener
        stt = SpeechRecognizer.createSpeechRecognizer(this);
        stt.setRecognitionListener(new BaseRecognitionListener(){
            public void onResults(Bundle results) {
                // Recupera as possíveis palavras que foram pronunciadas
                ArrayList<String> words = results.getStringArrayList(
                    SpeechRecognizer.RESULTS_RECOGNITION);
                listView.setAdapter(new ArrayAdapter<String>(getBaseContext(),
                    android.R.layout.simple_list_item_1, words));
            }
        });
        // Inicia o listener do reconhecimento de voz
        Intent intent = getRecognizerIntent();
        stt.startListening(intent);
    }
    protected Intent getRecognizerIntent() {
        Intent intent = new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
        // Mesma intent, simplifiquei o código aqui...
        intent.putExtra(RecognizerIntent.EXTRA_LANGUAGE, "pt-BR");
        return intent;
    }
    @Override
    protected void onDestroy() {
        super.onDestroy();
        // Libera os recursos e finaliza o STT
        stt.stopListening();
        stt.destroy();
    }
}
```

No layout desta activity adicione apenas um ListView:

 /res/layout/activity_hello_speech_recognizer.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout ... android:orientation="vertical" >
    <ListView android:id="@+id/list"
        android:layout_width="match_parent" android:layout_height="match_parent " />
</LinearLayout>
```

Apenas para constar, para esse código funcionar, é necessário adicionar a permissão `android.permission.RECORD_AUDIO` no arquivo de manifesto, mas já fizemos isso anteriormente ao utilizar o `synthesizeToFile(texto,params,arquivo)`, que converte o texto para voz e salva o áudio em arquivo.

A diferença deste exemplo para o anterior que utilizava intents era que no primeiro fizemos um botão que chamava a intent, ou seja, o aplicativo nativo de reconhecimento de voz. Desta vez, a activity inicia a engine de reconhecimento de voz em segundo plano e fica monitorando tudo o que é falado. Portanto, se você falar algo, os resultados serão entregues no método `onResults(bundle)` e mostrados no `ListView`. Faça o teste.

37.6 Links úteis

Neste capítulo estudamos os recursos de voz disponíveis na plataforma do Android. Seguem alguns links para continuar seus estudos.

- **Android Developers Blog – Introduction to TTS**

<http://android-developers.blogspot.com.br/2009/09/introduction-to-text-to-speech-in.html>

- **YouTube – Palestra sobre TTS no Google I/O 2009**

<https://www.youtube.com/watch?v=xS-ju61vOQw>



CAPÍTULO 38

Gradle

O Gradle (gradle.org) é um moderno sistema de builds que segundo a própria documentação oficial é definido com a seguinte frase: “Gradle combina o poder e a flexibilidade do Ant com o gerenciamento de dependência e convenções do Maven, em uma maneira mais eficaz”.

Neste capítulo vamos estudar alguns dos recursos mais interessantes do Gradle aplicados ao desenvolvimento Android, como gerenciamento de dependências e builds customizados.

38.1 Introdução

Ant (ant.apache.org) e Maven (maven.apache.org) são dois sistemas de builds bastante populares, e o Gradle (gradle.org) é a evolução que combina o melhor de cada um. Tanto o Ant quanto o Maven ou Gradle são assuntos que podem ser abordados em um livro inteiro, mas neste capítulo vamos estudar o básico sobre o Gradle e principalmente sobre como ele pode ajudar no desenvolvimento para Android.

A esta altura do livro, você já deve ter entendido que um dos principais benefícios do Gradle é gerenciar as dependências, pois por diversas vezes durante o livro atualizamos o arquivo *app/build.gradle* para declarar as bibliotecas necessárias. Para relembrar, as dependências são configuradas dentro da estrutura *dependencies*, conforme demonstrado a seguir:

```
 app/build.gradle

apply plugin: 'com.android.application'
...
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
}
```

```
compile 'com.android.support:appcompat-v7:22.1.0'  
compile 'br.com.livroandroid:android-utils:1.0.0'  
}
```

38.2 Gerenciando dependências

No Gradle podemos declarar módulos do projeto como dependências ou bibliotecas, sendo que as bibliotecas podem estar em um repositório local ou remoto.

Por padrão, todas as bibliotecas adicionadas no *build.gradle* buscam os artefatos no repositório do Maven JCenter, localizado no seguinte endereço:

<http://jcenter.bintray.com/>

Mesmo que você não declare o repositório JCenter no arquivo *build.gradle*, ele é usado como repositório padrão.

```
repositories {  
    jcenter()  
}
```

O JCenter é um superconjunto, ou seja, ele engloba o famoso Maven Central (<http://repo1.maven.org/maven2/>), outro gigantesco repositório de código.

```
repositories {  
    mavenCentral()  
}
```

Como eu disse antes, o repositório do JCenter é padrão e ele será utilizado para buscar as dependências mesmo que não esteja declarado no *build.gradle*.

Mas com certeza o que você ou sua empresa vai precisar no dia a dia é criar bibliotecas internas com classes utilitárias. Ao criar a biblioteca, você terá de decidir se ela será livre e de código aberto, como a biblioteca *android-utils* que fiz para o livro, ou se ela será privada, ou seja, somente você e o pessoal da sua empresa poderão utilizá-la.

Bibliotecas livres podem ser publicadas no repositório global do Maven Central ou JCenter, e foi exatamente o que fiz com a biblioteca *android-utils*.

No caso da minha biblioteca, eu a publiquei no Maven Central.

<http://repo1.maven.org/maven2/br.com/livroandroid/android-utils/>

E como o JCenter engloba o Maven Central, também podemos encontrar a lib aqui:

<http://jcenter.bintray.com/br.com/livroandroid/android-utils/>

Mas o que fazer caso a biblioteca precise ser privada da sua empresa? Nesse caso, elas podem ser publicadas em um repositório local do seu computador ou em algum servidor remoto interno da empresa.

Apenas para ficar claro o significado de cada tipo de repositório, veja este exemplo de arquivo *build.gradle*. Nele está declarado o repositório global do JCenter (já incluso por padrão), um repositório local do computador e um repositório remoto. No caso do repositório remoto é preciso instalar o servidor web Sonatype Nexus, que vamos estudar mais para frente.

```

 app/build.gradle

apply plugin: 'com.android.application'

...
dependencies { ... }
repositories {
    jcenter()
    maven { url "file:///c:/gradle/rep" }
    maven { url "http://localhost:8081/nexus/content/repositories/releases/" }
}

```

38.3 Trabalhando com módulos

Na documentação do Android é muito enfatizada a criação de módulos para conter o código comum do projeto, ou seja, as bibliotecas ou classes que você pode reutilizar entre vários módulos.

Para praticar, vamos fazer um exercício. Crie um projeto chamado **HelloModulo** com uma **MainActivity** padrão. Logo depois, utilize o wizard **File > New Module** e selecione a opção **Android Library**. Na próxima página do wizard digite **MyLibrary** para o nome do módulo, prossiga com o wizard e clique em **Finish**.

Feito isso, você terá o módulo **app**, que é padrão do Android Studio, e o módulo **MyLibrary**, que foi criado para ter classes utilitárias, conforme a figura 38.1. Veja que no módulo **MyLibrary** criei uma classe chamada **ClasseUtilitaria**, a qual pode ser compartilhada com o módulo principal, ou seja, o módulo **app**.

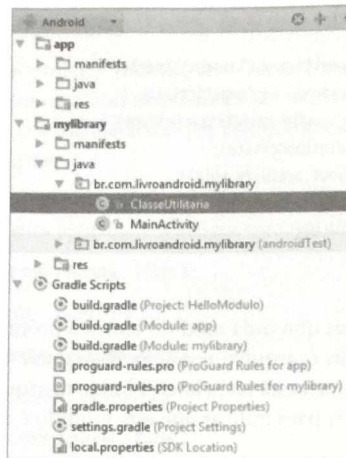


Figura 38.1 – Módulo criado.

Mas, para que a dependência entre os módulos funcione, precisamos configurar o arquivo `app/build.gradle` do módulo `app`, adicionando a dependência para o módulo `MyLibrary`, pois isso não é feito automaticamente.

`app/build.gradle`

```
apply plugin: 'com.android.application'

dependencies {
    compile fileTree(include: ['*.jar'], dir: 'libs')
    compile 'com.android.support:appcompat-v7:22.1.0'
    compile project(':mylibrary')
}
```

Dica: módulos são usados para conter o código referente ao Android Wear, TV, Glass etc., como também podem conter o código reutilizável para todos os módulos do projeto. Outro benefício dos módulos é que cada um pode ser compilado e executado separadamente.

É assim que dependências para módulos são adicionadas. Feito isso, sempre que compilar o módulo `app`, todas as classes do módulo `MyLibrary` serão adicionadas no build, ou seja, podemos utilizar as classes do módulo `MyLibrary` no módulo `app`. Para comprovar a teoria, altere o código-fonte da `MainActivity` do módulo `app` para utilizar a classe `ClasseUtilitaria` do módulo `MyLibrary`, conforme demonstrado a seguir.

**MainActivity.java**

```
import br.com.livroandroid.mylibrary.ClasseUtilitaria;
public class MainActivity extends AppCompatActivity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        String s = ClasseUtilitaria.hello();
        Log.d("livro", ">> " + s);
    }
}
```

Por último, vale ressaltar que cada módulo tem seu próprio arquivo *build.gradle*; mas tenha atenção, pois o arquivo *mylibrary/build.gradle* é diferente do arquivo *app/build.gradle*. A diferença está na primeira linha do arquivo, pois é declarado o plugin `com.android.library` para indicar que este módulo é uma biblioteca.

**mylibrary/build.gradle**

```
apply plugin: 'com.android.library'
// O restante é igual
```

Isso é possível graças ao conceito de plugins do Gradle. Note que o módulo *app* é sempre declarado com o plugin `com.android.application`, indicando que esse módulo é executável, ou seja, é possível clicar no botão **Run** do Android Studio.

**app/build.gradle**

```
apply plugin: 'com.android.application'
// O restante é igual
```

Esse conceito é importante, pois somente bibliotecas podem ser compiladas e disponibilizadas para outros projetos. Neste caso estamos compilando um módulo que está na mesma estrutura de diretórios do projeto. Mas em casos mais avançados a biblioteca pode ser compilada de forma independente e instalada em algum repositório, seja uma pasta local ou um servidor remoto.

Dica: módulos podem ser compilados e executados separadamente. Um exemplo disso são os módulos para Android Wear, Tablet e Glass que podemos criar no projeto. No botão **Run** do Android Studio podemos escolher qual o módulo que deve executar, desde que ele não seja uma biblioteca.

38.4 Trabalhando com bibliotecas

Caso você faça muitos projetos mobile, com certeza sentirá a necessidade de criar uma biblioteca para reutilizar suas classes em todos os seus aplicativos. Um exemplo de biblioteca é a `android-utils` que estudamos no livro; veja como ela facilitou os nossos estudos.

Caso tenha interesse em aprender a criar bibliotecas, como a `android-utils` que fiz para o livro, sugiro ler esta série de três artigos que publiquei no meu site.

- **Criando libs no Gradle com Android – Parte 1**

<http://ricardolecheta.com.br/?p=371>

- **Criando libs no Gradle com Android – Parte 2**

<http://ricardolecheta.com.br/?p=423>

- **Criando libs no Gradle com Android – Parte 3**

<http://ricardolecheta.com.br/?p=450>

A primeira parte do artigo mostra como criar um projeto biblioteca chamado **MyLib** e como fazer o build em uma pasta local do computador. Uma vez que o build é feito, o artigo mostra como declarar a dependência deste repositório local para compilar o projeto e utilizar a biblioteca. Farei um resumo da primeira parte do artigo no próximo tópico, mas as partes 2 e 3 estarão somente no site.

A segunda parte do artigo mostra como instalar e configurar o servidor web Sonatype Nexus, que é um servidor de dependências como o Maven. Veremos como fazer o build e enviar (upload) da biblioteca para o servidor.

A terceira parte deste artigo ensina como enviar uma biblioteca para o Maven Central, caso você queira distribuir sua biblioteca de forma global. Foi exatamente o que fiz com a biblioteca `android-utils` que usamos durante os estudos deste livro.

38.5 Criando uma biblioteca

Chega de teoria, vamos exercitar e aprender a criar uma biblioteca e publicá-la em um repositório local, ou seja, uma pasta no seu computador.

Para criar a biblioteca, crie um novo projeto no Android Studio chamado **MyLib**. No wizard nem precisa criar uma activity, pois não vamos usá-la. Na sequência, crie a classe `ToastUtil`. O objetivo dessa classe utilitária é facilitar o código para mostrar um toast na tela.

**ToastUtil.java**

```
public class ToastUtil {
    public static void toast(Context context, String msg) {
        Toast.makeText(context, msg, Toast.LENGTH_SHORT).show();
    }
}
```

Para transformar este projeto em biblioteca, precisamos aplicar o plugin `com.android.library` no arquivo `app/build.gradle`. Também vamos aproveitar e aplicar o plugin do `maven`, pois é necessário para fazer o build da biblioteca e instalá-la no repositório, seja este local ou remoto.

**app/build.gradle**

```
apply plugin: 'com.android.library'
apply plugin: 'maven'
android {
    compileSdkVersion 22
    buildToolsVersion "22.0.1"
    defaultConfig {
        // ATENÇÃO! FOI REMOVIDO applicationId
        minSdkVersion 9
        targetSdkVersion 22
        versionCode 1
        versionName "1.0"
    }
    buildTypes { // Tudo igual aqui... }
}
uploadArchives {
    repositories {
        mavenDeployer {
            repository(url: "file:///home/ricardo/gradle/rep")
            pom.groupId = "br.com.livroandroid"
            pom.artifactId = "mylib"
            pom.version = "0.0.1"
        }
    }
}
task install(dependsOn: uploadArchives)
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:22.1.0'
}
```

Atenção: uma biblioteca não pode declarar no arquivo *build.gradle* o item *applicationId*, o qual representa o pacote que identifica a aplicação.

No arquivo *build.gradle* da biblioteca foi criada a task *uploadArchives*, que define o repositório no qual a lib deve ser instalada. A última linha dessa configuração indica que a tarefa *install* depende da tarefa *uploadArchives*. Portanto, se você executar a task *install*, também será feito o upload da lib para o repositório, que neste caso é uma pasta local do computador. Dentro da tag *mavenDeployer* é definido o local do repositório para instalar a biblioteca, assim como o *groupId*, *artifactId* e versão.

```
mavenDeployer {  
    repository(url: "file:///home/ricardo/gradle/rep")  
    pom.groupId = "br.com.livroandroid"  
    pom.artifactId = "mylib"  
    pom.version = "0.0.1"  
}
```

Para utilizar a biblioteca, é preciso seguir o padrão *groupId:artifactId:version*, ou seja, no projeto cliente a biblioteca *mylib* deve ser declarada assim:

```
compile 'br.com.livroandroid:mylib:0.0.1'
```

Mas espere! Antes vamos aprender a fazer o build da biblioteca. Abra a janela **Gradle** que fica no lado direito do Android Studio, ou se preferir utilize o menu **View > Tool Windows > Gradle**. Essa janela, conforme a figura 38.2, mostra todas as tasks (tarefas) que você pode executar. Procure pela task *install* e dê um duplo clique.

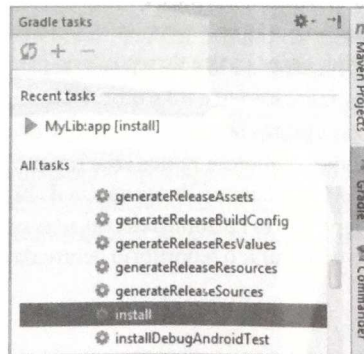


Figura 38.2 – Executando task *install*.

Ao executar a task `install`, será feita a compilação do projeto, e se tudo correr bem a biblioteca será publicada no repositório, que neste caso é uma pasta local. A figura 38.3 mostra o resultado dos logs na janela **Run** do Android Studio.

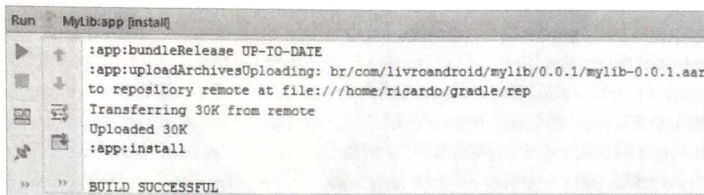


Figura 38.3 – Mensagens no console.

Para comprovar que a biblioteca foi realmente instalada no repositório local, a figura 38.4 mostra a pasta `file:///home/ricardo/gradle/rep` de meu computador. Naturalmente você pode alterar o caminho do repositório conforme preferir.



Figura 38.4 – Pasta do repositório com o build.

Nota: para testar o build, altere a pasta do repositório para alguma que exista em seu computador.

Pronto! Agora que já temos a biblioteca vamos criar um projeto para testá-la. Crie um projeto chamado **HelloLib** e adicione a dependência da biblioteca **MyLib**. Como a biblioteca está em um repositório customizado, ou seja, não está no JCenter ou Maven Central, é preciso configurar o repositório dentro da estrutura `repositories`.

```

app/build.gradle

apply plugin: 'com.android.application'
android { . . . }
repositories {
    maven { url "file:///home/ricardo/gradle/rep" }
  
```

Capítulo 38 ■ Gradle

```

}
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:22.1.0'
    compile 'br.com.livroandroid:mylib:0.0.1'
}

```

Uma vez que a biblioteca está declarada, podemos utilizar a classe `ToastUtil` no projeto:



MainActivity.java

```

import br.com.livroandroid.mylib.ToastUtil;
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ToastUtil.toast(this, "Teste Lib!");
    }
}

```

Do ponto de vista do projeto cliente, é isto; basta declarar a dependência e os repositórios em que estão as bibliotecas. O resto, o Gradle faz sozinho.

Apenas uma última dica que pode ser interessante. Digamos que você cometa um erro e precise lançar um novo release da sua biblioteca. Nesse caso se você incrementar a versão de 0.0.1 para 0.0.2 e fazer o build, basta também incrementar a versão para 0.0.2 no projeto cliente e tudo estará resolvido. Porém, muitas vezes queremos ajustar a versão atual. Ao fazer o build da mesma versão, neste caso da 0.0.1, ela será instalada no repositório normalmente. Mas o problema é que no projeto cliente do Android Studio a lib não será atualizada, pois o Gradle fez cache da biblioteca, sendo que ele já baixou esta versão no projeto.

O cache do Gradle fica na pasta `/Projeto/app/build/intermediates/exploded-aar`, conforme mostra a figura 38.5. Para limpar o cache, apague da biblioteca essa pasta; feito isso, no próximo build o Gradle vai sincronizar os arquivos e baixar novamente as dependências.

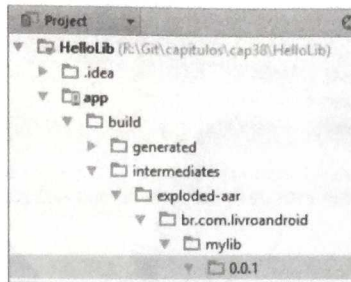


Figura 38.5 – Pasta de cache do Gradle.

38.6 Configurando um servidor Maven Sonatype Nexus

O servidor Sonatype Nexus pode ser instalado em qualquer servidor e é uma aplicação web. Ele foi criado pela Sonatype, empresa que também criou o Maven Central. Se você instalou o servidor Sonatype Nexus em algum computador, basta declarar a URL do repositório conforme demonstrado a seguir. O resto é tudo igual.

 **app/build.gradle**

```
apply plugin: 'com.android.application'
...
repositories {
    maven { url "http://servidor_da_sua_empresa:8081/nexus/content/repositories/releases/" }
}
```

Caso tenha interesse em aprender a configurar esse servidor, leia a parte 2 da série de artigos sobre como criar libs no Gradle (veja os links no último tópico).

38.7 Publicando no Maven Central

Publicar uma biblioteca no Maven Central é outra tarefa que muitos desenvolvedores Android precisam fazer. Contudo, isso é um tanto quanto complicado e exige diversos passos, principalmente da primeira vez.

Esse assunto está fora do escopo do livro, mas se tiver interesse recomendo ler a parte 3 do artigo sobre como criar libs no Gradle (veja os links no último tópico).

38.8 Flavors

Depois de estudar o conceito de dependências e bibliotecas, vamos aprender a utilizar flavors, recurso que permite criar diferentes builds do mesmo aplicativo.

Existem várias razões para fazer dois ou mais builds do mesmo aplicativo, e para exemplificar vou citar alguns exemplos:

- Criar uma versão free e uma paga do mesmo aplicativo.
- Criar uma versão de homologação e outra de produção do mesmo aplicativo. A versão de homologação utiliza os web services de homologação e vice-versa.
- Criar versões customizadas do mesmo aplicativo, trocando layouts e cores, assim como pequenas regras de negócios. Porém, no geral, é o mesmo aplicativo.

Enfim, não importa qual seja o seu objetivo, o Gradle pode ajudá-lo, e é justamente aqui que entram os flavors (sabores).

Para demonstrar o que é um flavor, vamos criar duas versões do aplicativo dos carros. A versão oficial terá as cores em azul, conforme já fizemos. Mas faremos uma versão plus que terá as cores em vermelho. Para isso, adicione os flavors azul e vermelho no arquivo *app/build.gradle*, conforme demonstrado a seguir. Se preferir, abra o projeto dos carros de exemplo deste capítulo.



app/build.gradle

```
apply plugin: 'com.android.application'
android {
    . . .
    defaultConfig { . . . }
    productFlavors {
        azul {
            applicationId "br.com.livroandroid.carros.azul"
        }
        vermelho {
            applicationId "br.com.livroandroid.carros.vermelho"
        }
    }
}
repositories { . . . }
dependencies { . . . }
```

Neste caso foram criados dois builds diferentes, ou seja, dois aplicativos diferentes cujos pacotes são: *br.com.livroandroid.carros.azul* e *br.com.livroandroid.carros.vermelho*.

Portanto, o usuário poderá ter instalado as duas versões do aplicativo ao mesmo tempo, pois deixei os flavors com pacotes diferentes.

Logo depois de criar um flavor, abra a janela **View > Tool Windows > Build Variants**, conforme mostra a figura 38.6. Build variant é uma combinação de flavors com build type.

Por padrão, o Android tem dois build types (tipos de build) que são: debug e release. O tipo de build debug é usado em desenvolvimento e assina a aplicação com o certificado debug.ksyore localizado na pasta do usuário no sistema operacional. Já o tipo de build release é utilizado para publicar o aplicativo na Google Play e também deve ser assinado, porém com outro certificado que você deve criar.

Então temos por padrão dois build types: debug e release. E agora no projeto dos carros temos dois flavors: azul e vermelho. Como uma build variant é uma combinação de flavors com build type, teremos quatro combinações que são: azulDebug, azulRelease, vermelhoDebug e vermelhoRelease (Figura 38.6). Portanto, ao executar o projeto, podemos deixar selecionada na janela **Build Variants** qual versão do build deve executar. Mas antes disso vamos aprender como customizar cada flavor.

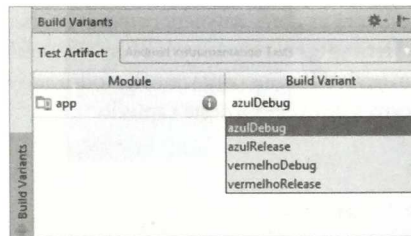


Figura 38.6 – Pasta de cache do Gradle.

Por padrão, o módulo app tem a seguinte estrutura de arquivos:

- `/app/src/main/res`
- `/app/src/main/java`
- `/app/src/main/AndroidManifest.xml`

Mas, como criamos dois flavors (sabores), podemos sobrescrever esses arquivos. Basta criar as pastas `/app/azul` e `/app/vermelho`, conforme demonstrado a seguir:

- `/app/src/azul/res`
- `/app/src/azul/java`
- `/app/src/azul/AndroidManifest.xml`

- `/app/src/vermelho/res`
- `/app/src/vermelho/java`
- `/app/src/vermelho/AndroidManifest.xml`

No caso do aplicativo dos carros, vamos criar apenas a pasta `/res` para sobrescrever os recursos, portanto crie as seguintes pastas no projeto:

- `/app/src/main/res` (esta já existe)
- `/app/src/azul/res`
- `/app/src/vermelho/res`

Agora vamos começar a customizar o aplicativo. Para começar com algo simples, vamos trocar o nome do aplicativo para **Carros Azul** ou **Carros Vermelho**, dependendo do tipo do build. Crie os seguintes arquivos (preste atenção no caminho dos arquivos).



`/app/src/azul/res/strings.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Carros Azul</string>
</resources>
```



`/app/src/vermelho/res/strings.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Carros Vermelho</string>
</resources>
```

A seguir, vamos customizar as cores do flavor **vermelho**. Note que no **azul** não vamos mexer, pois o padrão do aplicativo dos carros já é azul.



`/app/src/vermelho/res/colors.xml`

```
<resources>
    <color name="primary">#F44336</color> <!-- vermelho -->
    <color name="primary_dark">#C62828</color> <!-- vermelho escuro -->
    <color name="accent">#82FF59</color> <!-- verde -->
    <color name="control_highlight">#76FF03</color> <!-- verde -->
</resources>
```

Pronto, feito isso, as cores do tema Material para o flavor **vermelho** já estão todas modificadas. Por último, como o aplicativo dos carros tem um provedor de

conteúdo, precisamos customizá-lo no flavor vermelho, pois não é possível ter dois provedores de conteúdo com a mesma *authority*. No arquivo de manifesto do projeto dos carros, existe a seguinte configuração:



AndroidManifest.xml

```
...
<provider android:name=".domain.CarroContentProvider"
    android:authorities="@string/provider" android:exported="true"
    android:enabled="true"/>
```

Sendo assim, vamos customizar a *authority* do provedor de conteúdo para o flavor vermelho.



/app/src/vermelho/res/strings_config.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="provider">br.com.livroandroid.carrosvermelho</string>
</resources>
```

Lembrando que o arquivo *strings_config.xml* padrão está assim:



/app/src/main/res/values/strings_config.xml

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="provider">br.com.livroandroid.carros</string>
    <!-- Gerado do SHA1: C1:66:56:93:DF:DE:0B:B9:DC:ED:76:D7:65:B7:10:DC:1F:3F:0D:41 -->
    <string name="API_KEY">AIzaSyDYCu8-Ijlg-dug-VzT1SC_6fekDKn0oTQ</string>
</resources>
```

Pronto! Agora a customização chegou ao fim. Na janela **Build Variants** do Android Studio selecione o build azulDebug ou vermelhoDebug para executar o aplicativo. O resultado será o aplicativo dos carros nas cores azul e vermelha. Não mostrarei nenhuma figura com o resultado, pois não é possível ver as cores no livro, portanto execute o projeto no emulador.

A figura 38.7 mostra as duas aplicações instaladas no emulador, pois cada flavor foi configurado com um pacote diferente.

Dica: às vezes, trocar a build variant de azulDebug para vermelhoDebug não faz a compilação inteira do projeto. Portanto, se perceber que algo não foi compilado corretamente, force a compilação pelo menu Build > Rebuild Project.

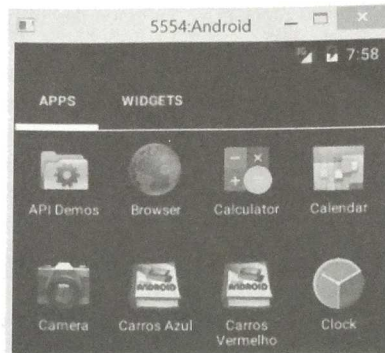


Figura 38.7 – Aplicativos instalados no emulador.

38.9 Classe BuildConfig

Se você ainda não conhece a classe `BuildConfig`, vamos apresentá-la agora, pois ela é gerada sempre que o projeto é compilado. O conceito é parecido com a classe `R`, a qual já é nossa velha conhecida.

O código-fonte a seguir mostra o resultado ao compilar o projeto em debug com o flavor vermelho. O código está comentado explicando cada constante disponível na classe `BuildConfig`.

 `app/build/.../BuildConfig.java`

```
public final class BuildConfig {
    // Indica se a aplicação foi compilada como Debug ou Release
    public static final boolean DEBUG = Boolean.parseBoolean("true");
    // Pacote da aplicação
    public static final String APPLICATION_ID = "br.com.livroandroid.carros.vermelho";
    // Indica o tipo de build
    public static final String BUILD_TYPE = "debug";
    // Indica o flavor (azul ou vermelho)
    public static final String FLAVOR = "vermelho";
    // Version code do build
    public static final int VERSION_CODE = 1;
    // Version name do build
    public static final String VERSION_NAME = "1.0";
}
```


Essas informações são muito úteis, pois podemos descobrir se a aplicação foi assinada com um certificado de debug ou release. Caso a aplicação seja assinada com um certificado de debug, sabemos que se trata do ambiente de desenvolvimento e podemos adicionar vários logs no aplicativo para auxiliar na depuração. Já em modo release, é recomendado que todos os logs sejam desabilitados antes de publicar no Google Play. Isso é preciso porque, caso o dispositivo seja plugado na USB, todos os logs aparecerão no LogCat. Portanto, tome cuidado para não expor informações demais do seu aplicativo em modo release.

Mas o que mais gosto da classe `BuildConfig` é a constante `BuildConfig.FLAVOR`, que retorna o flavor com o qual o aplicativo foi compilado. Usando essa constante, é possível testar se o build do aplicativo foi feito com a versão azul ou vermelha.



`CarroService.java`

```
public class CarroService {  
    ...  
    public static boolean isBuildAzul() {  
        return "azul".equals(BuildConfig.FLAVOR);  
    }  
    public static boolean isBuildVermelho() {  
        return "vermelho".equals(BuildConfig.FLAVOR);  
    }  
}
```

Assim, em qualquer lugar do aplicativo, é possível customizar a versão conforme o tipo do build:

```
if(CarroService.isBuildAzul()) {  
    // Build Azul  
} else if(CarroService.isBuildVermelho()) {  
    // Build Vermelho  
}
```

Dica: no aplicativo dos carros usamos o conceito de flavors para criar duas versões: azul e vermelho. Lembre-se de que esse recurso pode ser usado para várias situações diferentes, como criar versões gratuitas ou pagas do mesmo aplicativo, versões de homologação ou produção etc.

38.10 Assinando o aplicativo para o build release

Para publicar o aplicativo no Google Play, é necessário assiná-lo com um certificado de release, que é diferente do certificado de debug, o qual já explicamos por diversas vezes no livro.

O primeiro passo é criar o certificado de release. Isso pode ser feito abrindo um prompt e digitando a seguinte linha de comando com a ferramenta `keytool` disponível no JDK:

```
keytool -genkey -v -keystore carros.keystore -alias livro_android -keyalg RSA -validity 10000
```

Você terá de responder algumas perguntas ao digitar essa linha de comando. Mas eu particularmente prefiro criar o certificado utilizando o próprio Android Studio pelo menu **Build > Generate Signed APK**. No wizard, clique em **Create New** para criar um novo certificado. Na janela **New Key Store**, digite o nome do arquivo, senha do certificado, o alias, senha do alias e o restante das informações, conforme a figura 38.8.

Ao clicar em **OK**, o certificado (keystore) será gerado na pasta informada. Logo depois de gerar o certificado, o wizard fornece a opção para gerar um apk assinado com ele. Mas agora cancele o wizard, pois nosso objetivo era apenas gerar o certificado.

The image shows a 'New Key Store' dialog box with the following fields and values:

- Key store path: `n:\temp\carros.keystore`
- Password: `*****`
- Confirm: `*****`
- Key Alias: `carros`
- Password: `*****`
- Confirm: `*****`
- Validity (years): `25`
- Certificate information:
 - First and Last Name: `Livro Android`
 - Organizational Unit: `LivroAndroid`
 - Organization: `Livro Android`
 - City or Locality: `Curitiba`
 - State or Province: `PR`
 - Country Code (XX): `BR`

At the bottom are **OK** and **Cancel** buttons.

Figura 38.8 – Criando um certificado (keystore).

Agora copie o arquivo do certificado para a pasta `app` do projeto (Figura 38.9).

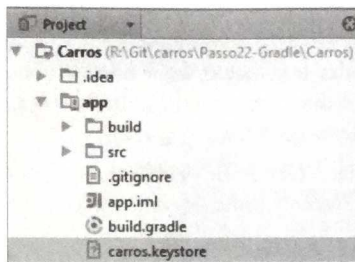


Figura 38.9 – Arquivo do certificado.

Como já temos o certificado, o próximo passo é configurar o Gradle para utilizá-lo ao fazer o build para **release**. Isso é feito no arquivo `build.gradle`. Basta adicionar a estrutura `signingConfigs` dentro da estrutura `android`, conforme mostrado a seguir. Também configure cada flavor para usar este item `signingConfig`. Tenha atenção, pois devemos informar o nome exato do arquivo keystore que foi criado, assim como a senha, o alias e a senha do alias.

 `app/build.gradle`

```
apply plugin: 'com.android.application'

android {
    . . .
    signingConfigs {
        release {
            storeFile file("carros.keystore")
            storePassword "carros"
            keyAlias "carros"
            keyPassword "carros"
        }
    }
    productFlavors {
        azul {
            applicationId "br.com.livroandroid.carros.azul"
            signingConfig signingConfigs.release
        }
        vermelho {
            applicationId "br.com.livroandroid.carros.vermelho"
            signingConfig signingConfigs.release
        }
    }
}

buildTypes { . . . }
```

```
}  
repositories { . . . }  
dependencies { . . . }
```

Dica: uma janela interessante do Android Studio que permite configurar o arquivo *build.gradle* visualmente é a File > Project Structure. Nessa janela, é possível visualizar o local do SDK, configurações do projeto e todas as informações de build do módulo *app*, como flavors, assinatura dos certificados e ainda gerenciar as dependências. Dê uma brincada nesta janela, pois tenho certeza de que você vai gostar. Ao alterar qualquer informação nesses formulários, tudo será refletido no arquivo *build.gradle*.

E como fazer o build para **release**? Isso é simples, basta selecionar uma das build variants *azulRelease* ou *vermelhoRelease* e executar o projeto.

O build final, ou seja, o arquivo *.apk*, fica na pasta *app/build/outputs* do projeto.

No caso do aplicativo dos carros, ao fazer o build com o certificado de **debug**, os arquivos se chamarão *app-azul-debug.apk* e *app-vermelho-debug.apk*. Ao fazer o build com o certificado de **release**, os arquivos se chamarão *app-azul-release.apk* e *app-vermelho-release.apk*.

Outra opção para exportar os arquivos *.apk* de release é utilizar o wizard **Build > Generate Signed APK**. Basta selecionar o certificado keystore de **release** e preencher o formulário. Acredito que isso é simples, portanto faça o teste você mesmo.

Importante: o arquivo *.apk* assinado com o certificado de **release** representa o build final da aplicação, e é exatamente esse arquivo que você publicará no Google Play. Nunca perca o certificado de **release**, pois se isso acontecer não será possível atualizar o aplicativo na loja.

38.11 Links úteis

Neste capítulo estudamos o básico sobre como utilizar o Gradle no Android Studio. Como o assunto é muito extenso, procurei abordar o essencial que acredito que você usará no seu dia a dia, como por exemplo: criar uma biblioteca e diferentes builds do aplicativo com flavors.

Para continuar seus estudos, seguem alguns links da documentação oficial e também a série de três artigos publicados em meu site.

- **Gradle Docs – Dependency Management/Repositories**

https://gradle.org/docs/current/userguide/dependency_management.html#sec:repositories

- **Android Tools – Build System Overview**

<https://developer.android.com/sdk/installing/studio-build.html>

- **Android Tools – Building and Running from Android Studio**

<https://developer.android.com/tools/building/building-studio.html>

- **Android Tools – Gradle Plugin User Guide**

<http://tools.android.com/tech-docs/new-build-system/user-guide>

- **Criando libs no Gradle com Android – Parte 1**

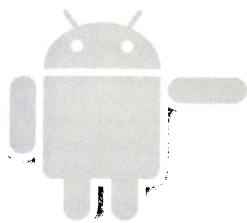
<http://ricardolecheta.com.br/?p=371>

- **Criando libs no Gradle com Android – Parte 2**

<http://ricardolecheta.com.br/?p=423>

- **Criando libs no Gradle com Android – Parte 3**

<http://ricardolecheta.com.br/?p=450>



CAPÍTULO 39

Android Wear

O Android Wear é o sistema operacional criado para wearables, como os relógios inteligentes (smartwatches). O termo *wearables* em português pode ser traduzido como dispositivos vestíveis.

Neste capítulo vamos aprender a integrar o aplicativo do smartphone com o relógio e desenvolver interfaces específicas para as telas pequenas do relógio.

39.1 Introdução

O Android Wear permite que você fique conectado ao seu smartphone e receba notificações em tempo real. Tem uma interface baseada em cards (cartões), de forma similar ao Google Now. A interação com o usuário é feita via gestos e comandos de voz.

A figura 39.1 mostra alguns dos relógios com Android Wear disponíveis no mercado na época que este livro estava sendo escrito. Em ordem da esquerda para direita estão: Samsung Gear Live, Moto 360 e LG G Watch.



Figura 39.1 – Relógios.

O mundo do desenvolvimento para Android é cheio de surpresas. Ora estamos desenvolvendo aplicativos para poderosos smartphones, ora estamos desenvolvendo

aplicativos para tablets nos quais podemos usufruir de um generoso espaço disponível na tela. A evolução da plataforma do Android não para, e com o surgimento dos relógios o foco é ter uma interface enxuta, simples e objetiva, específica para telas pequenas.

Criar interfaces para relógios é totalmente diferente de criar interfaces para smartphones e tablets, portanto precisamos primeiro entender como o relógio funciona e como deve ser a experiência do usuário neste tipo de dispositivo.

A lista a seguir mostra alguns dos conceitos sobre o design de interfaces para o wear:

- **Iniciar aplicativos ou cards automaticamente** – Usuários estão acostumados a iniciar os aplicativos manualmente, mas no wear muitas vezes os cards (cartões) simplesmente aparecem na tela, mostrando informações relevantes naquele momento: como um lembrete de uma reunião, alguma mensagem ou email, uma notificação etc.
- **Interfaces simples, práticas e rápidas (glanceable)** – A interface de usuário deve ser leve e prática, pois, quanto menos tempo o usuário levar para entender o significado da tela, melhor. O ideal é que o usuário bata o olho na tela e imediatamente consiga entender os dados apresentados. Por exemplo, cartões que mostram informações sobre o tempo ou informações sobre o horário de um voo são simples, práticos e concisos, mostram pouca informação, mas o suficiente; assim deve ser a interface para os relógios.
- **Sugestão sobre demanda** – Aplicativos para relógio são responsivos e podem reagir a eventos para auxiliar o usuário em momentos do dia a dia, como enviar uma mensagem de texto rapidamente para um contato, por meio de um comando de voz.
- **Interações simples** – A interação com o relógio é toda por meio de gestos e comandos por voz e deve ser o mais simples e prática possível.
- **Crie o design para telas pequenas** – Ao contrário da tela de um smartphone, o relógio deve mostrar poucas informações e com ícones grandes, para facilitar o toque nos elementos gráficos.
- **Utilize cartões (cards)** – Cartões promovem uma interface consistente em toda a plataforma do Android e representam um aspecto importante do Material Design. Cartões são simples, práticos e mostram rapidamente a informação para o usuário.
- **Respostas rápidas** – Facilite a interação do usuário com o aplicativo. Quanto mais rápido o usuário interagir e executar alguma ação, melhor.

Para mais detalhes sobre os princípios de design para o Android Wear, recomendo ler a documentação oficial. Por ora, essa introdução é suficiente, o próximo passo é colocar a mão na massa.

<https://developer.android.com/design/wear/index.html>

39.2 Hello World Wear

Acredito que a esta altura do livro você não tenha nenhum problema com relação a criar um emulador e com certeza já descobriu que é possível criar emuladores para smartphones e tablets, Google TV, Glass e wear.

Ao criar um emulador para o wear, você deve escolher entre o dispositivo com uma tela quadrada (square) ou redonda (round). O problema é que o espaçamento das telas quadradas e redondas são diferentes e, se aplicarmos o mesmo layout em ambas as telas, teremos resultados diferentes, conforme mostra a figura 39.2. A tela quadrada tem 280x280 pixels, e a tela redonda, 320x320 pixels. Embora a tela redonda seja maior, ela perde muito espaço nos cantos devido às margens.

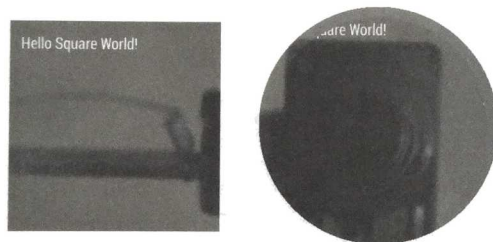


Figura 39.2 – Layout na tela quadrada e redonda.

Para solucionar esse problema, foi criada a biblioteca **Wearable UI Library**, que facilita criar interfaces para wearables. Então mãos à obra, faça os seguintes passos:

1. Crie dois emuladores para o wear: um quadrado, outro redondo.
2. Crie um novo projeto no Android Studio com suporte ao Android Wear (selecione o módulo do **wear** no wizard).
3. Execute o módulo **wear** do projeto criado nos dois emuladores do wear.

A figura 39.3 mostra o resultado do Hello World nos dois emuladores (rect e round). Observe que, como utilizamos o wizard do Android Studio, cada versão mostrou uma mensagem diferente.

Dica: ao executar o emulador do wear, siga as instruções para aprender a manusear os cards (cartões). O pequeno tutorial do emulador vai ajudá-lo nos primeiros passos. É bem divertido.



Figura 39.3 – Layout separado com *WatchViewStub*.

Por padrão, o wizard do Android Studio gera dois layouts distintos, um para a tela quadrada (square) e outro para a tela redonda (round). O layout da activity principal utiliza a view *WatchViewStub*, que permite configurar os layouts para cada tipo de tela com os atributos `app:rectLayout` e `app:roundLayout`. Conforme o tipo da tela do dispositivo, o *WatchViewStub* vai escolher o layout correto em tempo de execução.

```

wear /wear/res/layout/activity_main_wear.xml

<android.support.wearable.view.WatchViewStub
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/watch_view_stub"
    android:layout_width="match_parent" android:layout_height="match_parent"
    app:rectLayout="@layout/rect_activity_main_wear"
    app:roundLayout="@layout/round_activity_main_wear" />

```

Sendo assim, é possível ter dois layouts distintos para cada tipo de tela, quadrada ou redonda. A seguir podemos ver estes arquivos de layout.

```

wear /wear/res/layout/rect_activity_main_wear.xml

<LinearLayout android:orientation="vertical" . . . >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/hello_square" />
</LinearLayout>

```

```

wear/res/layout/round_activity_main_wear.xml
<LinearLayout android:orientation="vertical" . . . >
    <TextView android:id="@+id/text"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/hello_round" />
</LinearLayout>

```

No código da activity, a diferença é que não se pode utilizar o método `findViewById(id)` diretamente, é necessário obter a view do `WatchViewStub` que controla os dois layouts e utilizá-lo para obter as views internas:

```

MainWearActivity.java
public class MainWearActivity extends Activity {
    private TextView mTextView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main_wear);
        final WatchViewStub stub = (WatchViewStub) findViewById(R.id.watch_view_stub);
        stub.setOnLayoutInflatedListener(new WatchViewStub.OnLayoutInflatedListener() {
            @Override
            public void onLayoutInflated(WatchViewStub stub) {
                // Faça o findViewById com ajuda do WatchViewStub
                mTextView = (TextView) stub.findViewById(R.id.text);
            }
        });
    }
}

```

Pronto. Depois de obter as views, o restante do desenvolvimento é normal, como em qualquer outra activity. Lembrando que também podemos utilizar fragments no wear, inclusive existe o `CardFragment` que facilita mostrar um cartão na tela.

Antes de prosseguirmos, repare que o Android Studio adicionou as bibliotecas do wear no arquivo *build.gradle* do módulo **wear**:

```

wear/build.gradle
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    // Wearable UI Library
    compile 'com.google.android.support:wearable:1.1.0'
    // Google Play Services para wear
    compile 'com.google.android.gms:play-services-wearable:7.0.0'
}

```

Dica: o `WatchViewStub` permite utilizar layouts distintos para telas quadradas e redondas, assim podemos fazer customizações específicas para cada tipo de tela. Em tempo de execução, o layout correto será aplicado conforme a tela do dispositivo.

Outra forma de criar layouts é utilizar um layout único para ambas as telas, quadradas e redondas. Isso é possível com a classe `BoxInsetLayout`, que aplica espaçamentos no layout da tela redonda para que o conteúdo se encaixe melhor na tela. Para testar, altere o código-fonte do arquivo de layout principal conforme demonstrado a seguir:

```
 /wear/res/layout/activity_main_wear.xml

<?xml version="1.0" encoding="utf-8"?>
<android.support.wearable.view.BoxInsetLayout . . .
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="15dp">
    <LinearLayout android:orientation="vertical"
        android:layout_width="match_parent" android:layout_height="match_parent"
        android:padding="5dp" app:layout_box="all" >
        <TextView android:id="@+id/text" android:text="@string/hello_round"
            android:layout_width="wrap_content" android:layout_height="wrap_content" />
    </LinearLayout>
</android.support.wearable.view.BoxInsetLayout>
```

O importante deste layout é o atributo `android:padding="15dp"` do `BoxInsetLayout`, pois ele é aplicado apenas nas telas quadradas, para que o espaçamento fique igual à tela redonda. A vantagem de utilizar o `BoxInsetLayout` é que temos um layout único para todos os tipos da tela, e a programação no lado da activity volta ao normal, pois podemos usar o método `findViewById(id)` diretamente, sem aquele stub.

```
 MainWearActivity.java

public class MainWearActivity extends Activity {
    private TextView mTextView;
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main_wear);
        mTextView = (TextView)findViewById(R.id.text);
    }
}
```

Depois dessa alteração, execute o projeto novamente no emulador do wear, e o resultado deve ser o mesmo nas duas telas. Note que eu coloquei uma cor cinza no

fundo do layout, assim podemos ver que o `BoxInsetLayout` aplica um espaçamento no layout para que o resultado seja similar nos dois tipos de tela.

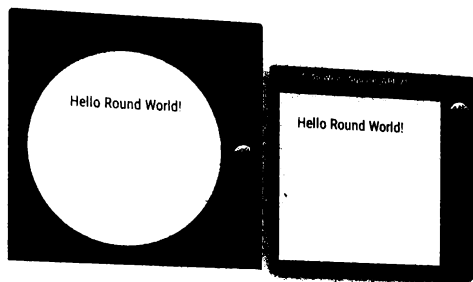


Figura 39.4 – Layout único com `BoxInsetLayout`.

Nota: o `WatchViewStub` utiliza layouts distintos para telas quadradas e redondas, já o `BoxInsetLayout` utiliza o mesmo layout, adicionando um espaçamento para deixar as telas parecidas. Você deverá escolher uma forma ou outra de fazer os layouts, conforme suas necessidades.

Para finalizar os conceitos básicos sobre criar interfaces e layouts para wear, vamos olhar como funciona os temas. No caso do wear, são utilizados os temas `Theme.DeviceDefault` ou `Theme.DeviceDefault.Light`, portanto se você olhar no arquivo de manifesto encontrará a configuração de tema da aplicação:

```
<manifest . . . >
  <application . . .
    android:theme="@android:style/Theme.DeviceDefault" >
    . . .
  </application>
</manifest>
```

39.3 Conectando o smartphone no Android Wear

A primeira tarefa que devemos fazer para desenvolver para wear é conectar o smartphone com o relógio. Para isso, você vai precisar de um smartphone (dispositivo real) com Android 4.3 ou superior com o Google Play Services instalado. No seu smartphone, faça o download do aplicativo do **Android Wear** disponível no Google Play.

Para conectar (parear) o smartphone com o emulador do relógio, conecte o smartphone na USB e digite o seguinte comando no prompt:

```
adb -d forward tcp:5601 tcp:5601
```

Importante: sempre que conectar o smartphone à USB ou reiniciar o emulador do wear, é necessário executar o comando `adb -d forward tcp:5601 tcp:5601` para parear os dispositivos.

Feito isso, abra o aplicativo do **Android Wear** no smartphone e utilize o menu **Conectar Emulador (Connect Emulator)** para fazer a conexão com o emulador. A figura 39.5 mostra o aplicativo do **Android Wear** conectado no emulador do wear.

A comunicação entre os dispositivos é feita com Bluetooth, portanto certifique-se de que o Bluetooth do smartphone esteja ligado. Isso significa que, para os dispositivos estarem conectados (pareados), eles precisam estar próximos e no raio de ação do Bluetooth, que é algo em torno de no máximo 8 a 10 metros.

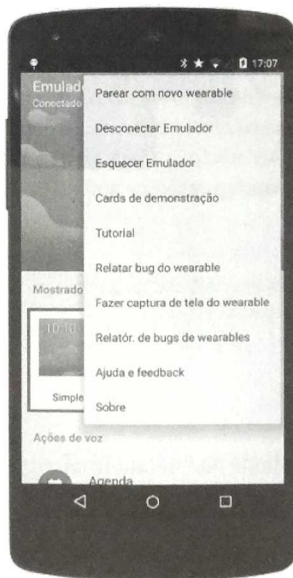


Figura 39.5 – Aplicativo do Android Wear no smartphone.

Dica: ao conectar o smartphone no wear, todas as notificações criadas no smartphone serão mostradas no relógio. Isso é o padrão. Recomendo você clicar no menu **Cards de Demonstração (Demo Cards)** no aplicativo do Android Wear para disparar vários exemplos de notificações para o emulador do wear. Assim você já terá uma boa ideia do tipo de interface recomendada para o relógio.

39.4 Conectando o smartphone no relógio físico

Caso você tenha um relógio com o Android Wear e queira utilizá-lo no lugar do emulador, siga os seguintes passos para conectá-lo ao seu smartphone:

1. No relógio, entre na tela de **Configurações > Sobre (Settings > About)** e clique no item **Número do Build (Build number)** por sete vezes. Isso vai habilitar as opções de desenvolvedor.
2. Entre no menu das **Opções do Desenvolvedor (Developer Options)** e habilite o debug pela USB.
3. Conecte o relógio na USB do computador. Ao fazer isso, uma mensagem será exibida em ambos os dispositivos, solicitando uma autorização para fazer o debug.
4. No aplicativo do Android Wear, clique no ícone de uma caixa de ferramentas (configurações) e selecione a opção **Depuração por Bluetooth**. Inicialmente a tela vai mostrar que os dispositivos estão desconectados, com as mensagens: Host: disconnected / Target: connected.

5. Abra um prompt e digite os seguintes comandos:

```
adb forward tcp:4444 localabstract:/adb-hub
```

```
adb connect localhost:4444
```

6. Na tela de configurações do aplicativo do Android Wear você verá que os dispositivos foram conectados com as mensagens Host: connected / Target: connected.

Para mais informações, consulte o tópico **Debugging over Bluetooth** da documentação oficial:

<https://developer.android.com/training/wearables/apps/bt-debugging.html>

De qualquer forma, recomendo utilizar o emulador do wear para testar os exemplos do livro, pois todos funcionam no emulador.

39.5 Notificações no wear

Com o relógio conectado ao smartphone, todas as notificações do smartphone aparecem no relógio. Mas, caso você queira, é possível pela API configurar uma notificação para apenas aparecer no smartphone ou relógio.

Para brincar com as notificações, abra o projeto **HelloNotification** que fizemos no capítulo 25, sobre notificações, e dispare as notificações no smartphone (estando conectado no emulador do wear). Ao fazer isso, todas as notificações serão mostradas no smartphone e no emulador do wear.

A figura 396 mostra a notificação simples do projeto **HelloNotification** no emulador do wear. No lado esquerdo da figura, a notificação aparece na tela inicial e ao selecioná-la o cartão (card) sobe para cima (parte 2). Na terceira parte da figura, podemos ver a página que mostra a opção para abrir a notificação no smartphone. Sempre que uma intent estiver associada à notificação, a opção **Open on phone** estará disponível.



Figura 396 – Notificação no wear.

Outro exemplo que temos no projeto **HelloNotification** é a notificação grande (big notification) na qual adicionamos três mensagens. A figura 397 mostra essa notificação no emulador do wear.

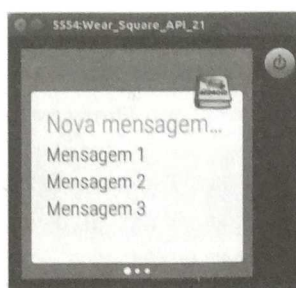


Figura 397 – Notificação grande no wear.

É por último um dos exemplos mais interessantes, o qual também fizemos no capítulo 25, sobre notificações, é criar uma notificação com ações. A figura 39.8 mostra a notificação com a ação de **pause** e **play**. Ao executar essas ações, intents serão disparadas no aplicativo. Esse tipo de notificação é muito útil para que o usuário selecione uma opção para responder alguma informação de forma rápida diretamente do seu relógio, sem tirar o celular do bolso.

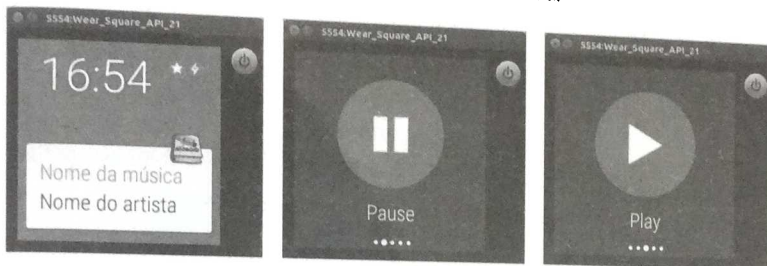


Figura 39.8 – Notificação no Wear.

39.6 Notificações com várias páginas

No wear é muito comum termos notificações com várias páginas, de forma que o usuário pode fazer o scroll lateral para visualizar os próximos cartões (cards).

Para demonstrar como criar uma notificação com páginas, crie a classe `NotificationWearUtil` no módulo do smartphone, conforme demonstrado a seguir:

`NotificationWearUtil.java`

```
public class NotificationWearUtil {
    public static void createPagesNotification(Context context) {
        // Página 1
        NotificationCompat.Builder notificationBuilder =
            new NotificationCompat.Builder(context)
                .setSmallIcon(R.mipmap.ic_launcher)
                .setContentTitle("Página 1")
                .setContentText("Primeira mensagem")
                /*.setContentIntent(viewPendingIntent)*/;

        // Página 2
        Notification page2 =
            new NotificationCompat.Builder(context)
                .setSmallIcon(R.mipmap.ic_launcher)
                .setContentTitle("Página 2")
```



```

        .setContentText("Segunda mensagem")
        .build();
    // Cria as páginas
    Notification notification =
        notificationBuilder.extend(new NotificationCompat.WearableExtender()
            .addPage(page2)).build();
    // Dispara a notificação
    NotificationManagerCompat nm = NotificationManagerCompat.from(context);
    nm.notify(1, notification);
}
}

```

Nota: o importante desse código é a classe `WearableExtender`. Ela é responsável por adicionar funcionalidades específicas do wear nas notificações.

Em qualquer lugar do código (do smartphone), chame este método para criar a notificação:

```
NotificationWearUtil.createPagesNotification(this);
```

No smartphone somente a primeira notificação será exibida, porém no wear você verá as duas páginas conforme a figura 39.9. Se quiser conferir o exemplo funcionando, abra o projeto **HelloWearNotifications** no Android Studio. Lembre-se de executar o módulo **mobile** no smartphone e o módulo **wear** no emulador do Android Wear.

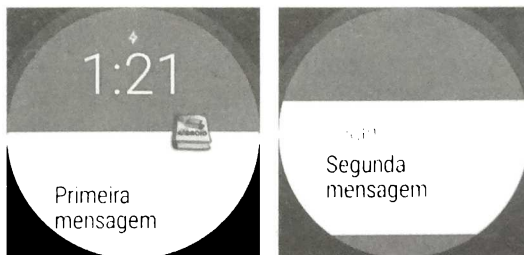


Figura 39.9 – Notificação com duas páginas.

39.7 Notificações empilhadas

Outro tipo de notificação muito comum no wear são as notificações empilhadas (stack notifications), que mostram uma notificação com um indicador de que existem mais N mensagens, conforme a figura 39.10.

No smartphone você verá estas três notificações separadamente, mas no wear automaticamente elas são agrupadas. Você também pode conferir este exemplo no projeto **HelloWearNotifications**.

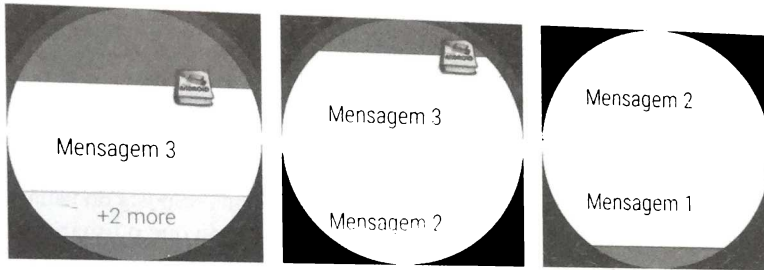


Figura 39.10– Notificações empilhadas.

A classe `NotificationUtil` da biblioteca `android-utils` contém um método `createStackNotification(...)` que cria uma notificação agrupada. O código é o mesmo utilizado para criar qualquer outra notificação, com exceção do método `setGroup(groupId)`, que recebe o código do grupo. Notificações que forem criadas com o mesmo código de grupo serão agrupadas automaticamente no wear.



NotificationUtil.java

```
public class NotificationUtil {
    . . .
    public static void createStackNotification(Context context, int id,String groupId,
        Intent intent,int smallIcon, String contentTitle, String contentText) {
        . . .
        // Cria a notification
        NotificationCompat.Builder builder = new NotificationCompat.Builder(context)
            .setContentIntent(p).setContentTitle(contentTitle)
            .setContentText(contentText).setSmallIcon(smallIcon)
            .setGroup(groupId)
            .setAutoCancel(true);
    }
}
```

Em qualquer lugar do código (do smartphone), chame este método para criar a notificação. Veja que estou passando sempre o mesmo código do grupo.

```
NotificationUtil.createStackNotification(this,1,"GrupoXXX",intent,R.mipmap.ic_launcher,
    "Título 1", "Mensagem 1");
```

```
NotificationUtil.createStackNotification(this,2,"GrupoXXX",intent,R.mipmap.ic_launcher,
    "Titulo 2","Mensagem 2");
NotificationUtil.createStackNotification(this,3,"GrupoXXX",intent,R.mipmap.ic_launcher,
    "Titulo 3","Mensagem 3");
```

39.8 Notificações com comandos de voz

Sem dúvida, os comandos de voz são uma das funções mais importantes do wear, e esse tipo de recurso pode ser facilmente criado por meio de uma notificação. Por exemplo, se o aplicativo do smartphone precisa coletar uma resposta do usuário, podemos enviar uma notificação para o relógio, solicitando que o usuário fale uma resposta ou escolha uma dentre as possíveis respostas apresentadas.

Para criar uma notificação que responda por meio de um comando de voz, é preciso criar um objeto do tipo `RemoteInput` informando um array de opções com as respostas (opcional) e a chave que será utilizada para enviar a resposta pela intent de retorno. No código de exemplo a seguir, a chave é a string `remote.input.key`.



NotificationWearUtil.java

```
public class NotificationWearUtil {
    public static void createRemoteInputNotification(Context context, Intent replyIntent) {
        // Intent para executar no smartphone ao responder
        PendingIntent replyPendingIntent =
            PendingIntent.getActivity(context, 0, replyIntent,
                PendingIntent.FLAG_UPDATE_CURRENT);

        // Remote Input
        String[] replyChoices = context.getResources().getStringArray(R.array.reply_choices);
        RemoteInput remoteInput = new RemoteInput.Builder("remote.input.key")
            .setLabel("Resposta") // Título
            .setChoices(replyChoices) // Array com respostas
            .build();

        NotificationCompat.Action action =
            new NotificationCompat.Action.Builder(R.drawable.ic_action_reply,
                "Responder", replyPendingIntent)
                .addRemoteInput(remoteInput)
                .build();

        // Cria a notificação
        Notification notification =
            new NotificationCompat.Builder(context)
                .setSmallIcon(R.mipmap.ic_launcher)
                .setContentTitle("Lembrete")
```

```

        .setContentText("Você vai à festa?")
        .extend(new NotificationCompat.WearableExtender().addAction(action))
        .build();
    NotificationManagerCompat notificationManager =
        NotificationManagerCompat.from(context);
    notificationManager.notify(1, notification);
}
}

```

O array com as respostas é definido no arquivo *arrays.xml*. Fornecer a lista com as possíveis respostas é opcional, porém é uma boa prática.

 /mobile/res/values/arrays.xml

```

<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="reply_choices">
        <item>Sim</item> <item>Não</item> <item>Talvez</item>
    </string-array>
</resources>

```

Para testar as notificações com resposta por comandos de voz, basta chamar um código como este e definir a activity que vai receber a resposta:

```

Intent intent = new Intent(this, ReplyActivity.class);
NotificationWearUtil.createRemoteInputNotification(this, intent);

```

Neste caso, quando o usuário responder o comando de voz no relógio, a activity *ReplyActivity* será chamada no smartphone para receber a resposta. Para ler a resposta, deve-se utilizar o método *RemoteInput.getResultsFromIntent(intent)* a fim de obter o *Bundle* e na sequência ler a mesma chave que foi utilizada para criar a notificação, que lembrando é a string *remote.input.key*.

 **ReplyActivity.java**

```

public class ReplyActivity extends ActionBarActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_reply);
        getSupportActionBar().setDisplayHomeAsUpEnabled(true);
        TextView text = (TextView) findViewById(R.id.text);
        text.setText(getMessageText(getIntent()));
        NotificationUtil.cancelAll(this);
    }
}

```

```
// Lê a resposta da voice input
private String getMessageText(Intent intent) {
    Bundle remoteInput = RemoteInput.getResultsFromIntent(intent);
    if (remoteInput != null) {
        // Mesma chave utilizada para criar a intent voice input
        CharSequence c = remoteInput.getCharSequence("remote.input.key");
        return c != null ? c.toString() : null;
    }
    return null;
}
}
```

Nota: notificações que respondem por comandos de voz são chamadas de Remote Input Notifications.

Caso queira conferir o resultado, este exemplo também está disponível no projeto **HelloWearNotifications**. A figura 39.11 mostra o resultado da notificação. Ao rolar a tela para o lado, o usuário pode clicar no botão **Responder**. Para responder, é possível utilizar o comando de voz caso você tenha um relógio real, ou escolher uma das opções que são previamente apresentadas na lista. Ao escolher a resposta, a activity recebe o retorno pela intent, mas isso você verá por si mesmo ao executar o exemplo.

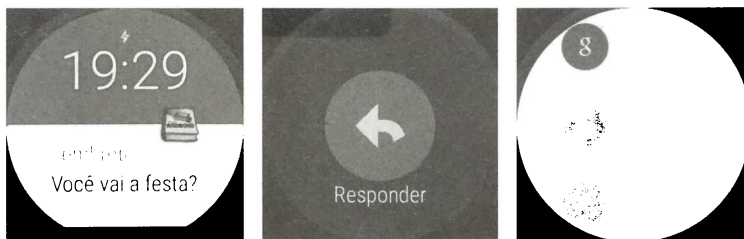


Figura 39.11 – Remote Input Notification.

39.9 Google Play Services e Wearable API

Uma das funcionalidades mais importantes do wear é a capacidade de enviar e receber mensagens do smartphone. Novamente, tudo funciona com base no Google Play Services, conforme mostra o seguinte template de código:

```

GoogleApiClient mGoogleApiClient = new GoogleApiClient.Builder(this)
    .addConnectionCallbacks(new ConnectionCallbacks() {
        public void onConnected(Bundle connectionHint) { . . . }
        public void onConnectionSuspended(int cause) { . . . } })
    .addOnConnectionFailedListener(new OnConnectionFailedListener() {
        public void onConnectionFailed(ConnectionResult result) { . . . }; })
    // Adiciona a Wearable API
    .addApi(Wearable.API)
    .build();

```

Como já estudamos o Google Play Services no livro, acredito que este código lhe deve ser familiar. Apenas observe que estamos adicionando a **Wearable API** no método `addAPI(api)`. Lembre-se também de que a dependência do Google Play Services deve ser declarada no arquivo *build.gradle* do módulo. Uma vez que a conexão com o Google Play Services foi estabelecida, podemos usar as bibliotecas de comunicação da **Wearable API**, a qual é dividida em três APIs que são: Node API, Message API e Data API.

39.10 Node API

A Node API é utilizada para obter o identificador do dispositivo conectado, seja o smartphone ou wearable. Esse identificador é chamado de `nodeId`.

Uma forma de obter o `nodeId` é utilizando o método `getConnectedNodes(...)`, conforme demonstrado a seguir. Note que é necessário informar uma implementação da interface `ResultCallback` para obter a resposta de forma assíncrona.

```

Wearable.NodeApi.getConnectedNodes(mGoogleApiClient).setResultCallback(
    new ResultCallback<NodeApi.GetConnectedNodesResult>() {
        @Override
        public void onResult(NodeApi.GetConnectedNodesResult getConnectedNodesResult) {
            List<Node> nodes = getConnectedNodesResult.getNodes();
            if (nodes != null && !nodes.isEmpty()) {
                Node node = nodes.get(0);
                nodeId = node.getId(); // Este é o nodeId
            }
        }
    });

```

Embora esse método retorne mais de um dispositivo conectado, podemos dizer que na maioria das vezes será apenas um. De qualquer forma, em aplicativos profissionais isso deve ser tratado. Outro método interessante da Node API é o `listener` que monitora se o dispositivo está conectado ou não. Para ativar o `listener`, basta utilizar este código:

```

GoogleApiClient mGoogleApiClient = . . .;
NodeApi.NodeListener nodeListener = . . .;
Wearable.NodeApi.addListener(mGoogleApiClient, nodeListener);

```

Como parâmetro deve-se informar uma implementação da interface `NodeApi.NodeListener` que contém dois métodos para indicar se o dispositivo foi conectado ou desconectado.

```

void onPeerConnected(Node node) { // dispositivo conectado }
void onPeerDisconnected(Node node) { // dispositivo desconectado }

```

Isso é muito importante para saber se o wearable está no raio de ação do smartphone, pois sabemos que a conexão entre eles é feita via Bluetooth.

39.11 Message API

A Message API é extremamente útil para enviar mensagens curtas e pequenas entre os dois dispositivos. A mensagem é identificada por um path (caminho) que deve começar com "/" e o conteúdo é um array de bytes.

Para enviar a mensagem, é necessário o `nodeId` obtido com a Node API. A seguir, temos um exemplo de mensagem enviada, no qual o path é `"/msg"`.

```

Wearable.MessageApi.sendMessage(mGoogleApiClient, nodeId, "/msg", new byte[]{1,2,3});

```

Para receber a mensagem, o dispositivo deve ativar o listener da Message API, informando uma implementação da interface `MessageApi.MessageListener`.

```

GoogleApiClient mGoogleApiClient = . . .;
MessageApi.MessageListener messageListener = . . .;
Wearable.DataApi.addListener(mGoogleApiClient, messageListener);

```

A interface `MessageApi.MessageListener` contém apenas o método `onMessageReceived(messageEvent)` que recebe a mensagem enviada. Neste método podemos obter o path que identifica a mensagem, o `nodeId` de origem e o conteúdo que é o array de bytes.

```

public void onMessageReceived(final MessageEvent messageEvent) {
    String path = messageEvent.getPath(); // identificador
    String nodeId = messageEvent.getSourceNodeId();
    byte[] bytes = messageEvent.getData(); // mensagem
}

```

Geralmente, as mensagens enviadas são pequenos bytes ou números inteiros que apenas indicam ações que os aplicativos devem fazer. Por exemplo, uma ação seria

“abra a activity x”, “mostre a notificação y” etc. A Message API foi criada para trocar informações de forma rápida entre os dispositivos.

Nota: a Message API é online, ou seja, é necessário que os dois dispositivos estejam conectados.

39.12 Data API

A Data API é útil para compartilhar informações entre o smartphone e o wearable utilizando uma estrutura simples de chave e valor. Os dados adicionados na estrutura serão automaticamente sincronizados entre o smartphone e o wearable. Mesmo se os dispositivos estejam desconectados, a sincronização será feita ao conectá-los.

Para enviar e armazenar informações na área de compartilhamento com a Data API, podemos utilizar um código como este:

```
Bundle bundle = new Bundle();
bundle.putString("nome", "Ricardo");
String path = "/msg";
PutDataMapRequest putDataMapReq = PutDataMapRequest.create(path);
DataMap dataMap = DataMap.fromBundle(bundle);
putDataMapReq.getDataMap().putAll(dataMap);
PutDataRequest putDataReq = putDataMapReq.asPutDataRequest();
Wearable.DataApi.putDataItem(mGoogleApiClient, putDataReq);
```

O identificador da mensagem é o caminho (path) `"/msg"`, e os dados podem ser enviados por um `Bundle`, o qual é convertido para um `DataMap`. Dessa forma, conseguimos compartilhar o conteúdo entre o smartphone e o wearable com a estrutura de chave e valor com a qual já estamos acostumados.

Do outro lado, para ler os dados, é preciso ativar o listener da Data API, para ser notificado sempre que ocorra uma mudança nos dados compartilhados entre os dispositivos.

```
GoogleApiClient mGoogleApiClient = . . .;
DataApi.DataListener dataListener = . . .;
Wearable.DataApi.addListener(mGoogleApiClient, dataListener);
```

A interface `DataApi.DataListener` contém apenas o método `onDataChanged(dataEventBuffer)`.

```
public void onDataChanged(DataEventBuffer dataEventBuffer) {
    for (DataEvent event : dataEventBuffer) {
        if (event.getType() == DataEvent.TYPE_CHANGED) {
```



```

        DataItem item = event.getDataItem();
        if (item.getUri().getPath().compareTo("/msg") == 0) {
            DataMap dataMap = DataMapItem.fromDataItem(item).getDataMap();
            String nome = dataMap.getString("nome");
        } else if (event.getType() == DataEvent.TYPE_DELETED) { . . . }
    }
}

```

Esse código verifica se o evento gerado é do tipo `DataEvent.TYPE_CHANGED`, para depois obter o objeto `DataItem` que contém os dados e um `DataMap` que funciona da mesma forma que um `Bundle`. Vale lembrar que a Data API pode ser usada pelos dois lados, sendo que ambos, `smartphone` e `wearable`, podem armazenar e ler as informações.

Nota: a Data API funciona de forma `offline`. Caso os dispositivos não estejam conectados, o evento `onDataChanged(...)` será chamado futuramente quando houver uma conexão. Recomenda-se que o conteúdo compartilhado não exceda 100kb, pois a transmissão é feita via Bluetooth.

39.13 Enviando mensagens entre o `smartphone` e o `Wear`

Agora vamos juntar os pedaços e criar um exemplo que vai unir a `Node API`, `Message API` e `Data API`.

Crie um projeto no Android Studio chamado **HelloWear** e no wizard selecione para criar os módulos para `smartphone/tablet` e `wear`. Siga o wizard passo a passo e crie a `MainActivity` de cada módulo. Por convenção, gosto de chamar a activity principal do `smartphone` de `MainMobileActivity` e a activity principal do `wear` de `MainWearActivity`.

Com o projeto criado, vamos criar um módulo para compartilhar o código-fonte entre os módulos `mobile` e `wear`. Crie um módulo com o wizard **File > New Module**, selecione o item **Android Library** e digite `shared` para o nome do módulo.

O resultado do projeto no Android Studio pode ser visto na figura 39.12. Ambos os módulos, `mobile` e `wear`, foram criados automaticamente pelo Android Studio. Dentro deles temos as classes e recursos com os quais já estamos acostumados. Lembre-se também de que cada módulo tem seu próprio arquivo `build.gradle` e pode ser executado separadamente pelo botão **Run** do Android Studio. A figura 39.12 também mostra o módulo `shared` que acabamos de criar. Dentro do módulo `shared`, vamos adicionar a classe `WearUtil` que será compartilhada pelos módulos `mobile` e `wear`.

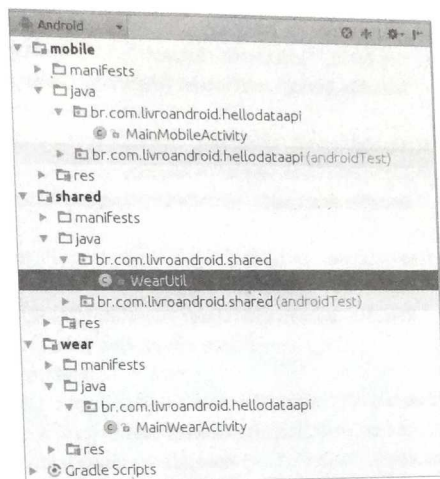


Figura 39.12 – Projeto com três módulos.

Conforme apresentado na figura 39.12, crie a classe `WearUtil` no módulo `shared`.

`WearUtil.java`

```
package br.com.livroandroid.shared;

...

public class WearUtil {
    private static final String TAG = "wear";
    private final GoogleApiClient mGoogleApiClient;
    private DataApi.DataListener dataListener;
    private MessageApi.MessageListener messageListener;
    private NodeApi.NodeListener nodeListener;
    // id do outro device (mobile ou wear)
    private String nodeId;

    public WearUtil(Context context) {
        this.dataListener = dataListener;
        this.messageListener = messageListener;
        mGoogleApiClient = new GoogleApiClient.Builder(context)
            .addConnectionCallbacks(new GoogleApiClient.ConnectionCallbacks() {
                @Override
                public void onConnected(Bundle connectionHint) {
                    findDeviceNodeId();
                    // Ao conectar, liga a DataAPI e MessageAPI
                    Log.d(TAG, "Play Services onConnected!");
                }
            })
    }
}
```

```

        if(dataListener != null) {
            Log.d(TAG, "addListener DataApi");
            Wearable.DataApi.addListener(mGoogleApiClient, dataListener);
        }
        if(messageListener != null) {
            Log.d(TAG, "addListener MessageApi");
            Wearable.MessageApi.addListener(mGoogleApiClient, messageListener);
        }
        if(nodeListener != null) {
            Log.d(TAG, "addListener NodeApi");
            Wearable.NodeApi.addListener(mGoogleApiClient, nodeListener);
        }
    }

    @Override
    public void onConnectionSuspended(int cause) {
        Log.d(TAG, "onConnectionSuspended: " + cause);
    }

})
.addOnConnectionFailedListener(new GoogleApiClient.OnConnectionFailedListener() {
    @Override
    public void onConnectionFailed(ConnectionResult result) {
        Log.d(TAG, "onConnectionFailed: " + result);
    }
})
.addApi(Wearable.API)
.build();
}

public void setDataListener(DataApi.DataListener dataListener) {
    this.dataListener = dataListener;
}

public void setMessageListener(MessageApi.MessageListener messageListener) {
    this.messageListener = messageListener;
}

public void setNodeListener(NodeApi.NodeListener nodeListener) {
    this.nodeListener = nodeListener;
}

// Connect to Google Play Services
public void connect() {
    Log.d(TAG, "connect()");
    mGoogleApiClient.connect();
}

// Reconnect to Google Play Services

```

```

public void disconnect() {
    Log.d(TAG, "disconnect()");
    if(mGoogleApiClient.isConnected()) {
        // Desliga as APIs do wear
        Wearable.DataApi.removeListener(mGoogleApiClient, this.dataListener);
        Wearable.MessageApi.removeListener(mGoogleApiClient, this.messageListener);
        Wearable.NodeApi.removeListener(mGoogleApiClient, this.nodeListener);
    }
    mGoogleApiClient.disconnect();
}

// DataAPI: Compartilha os dados
public void putData(String path, Bundle bundle) {
    Log.d(TAG, ">> putData() " + path);
    PutDataMapRequest putDataMapReq = PutDataMapRequest.create(path);
    DataMap dataMap = DataMap.fromBundle(bundle);
    putDataMapReq.getDataMap().putAll(dataMap);
    PutDataRequest putDataReq = putDataMapReq.asPutDataRequest();
    Wearable.DataApi.putDataItem(mGoogleApiClient, putDataReq);
}

// Message API: Envia uma mensagem
public void sendMessage(String path, byte[] msg) {
    Log.d(TAG, ">> sendMessage() " + path + ", to: " + nodeId);
    if(nodeId != null) {
        Wearable.MessageApi.sendMessage(mGoogleApiClient, nodeId, path, msg);
    }
}

// Node API descobre o id do outro device (mobile ou wear)
public void findDeviceNodeId() {
    // Chamada assíncrona, informa o callback
    Wearable.NodeApi.getConnectedNodes(mGoogleApiClient).setResultCallback(
        new ResultCallback<NodeApi.GetConnectedNodesResult>() {
            @Override
            public void onResult(NodeApi.GetConnectedNodesResult getConnectedNodesResult) {
                List<Node> nodes = getConnectedNodesResult.getNodes();
                if (nodes != null && !nodes.isEmpty()) {
                    Node node = nodes.get(0);
                    nodeId = node.getId();
                    Log.d(TAG, "findDeviceNodeId nodeId: " + nodeId);
                }
            }
        }
    );
}

```

```
// Cria um Asset a partir de um Bitmap
public Asset getAssetFromBitmap(Bitmap bitmap) {
    final ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
    bitmap.compress(Bitmap.CompressFormat.PNG, 100, byteStream);
    return Asset.createFromBytes(byteStream.toByteArray());
}

public Bitmap getBitmapFromAsset(Asset asset) {
    if (asset == null) { return null; }
    InputStream in = Wearable.DataApi.getFdForAsset(
        mGoogleApiClient, asset).await().getInputStream();
    Bitmap bitmap = BitmapFactory.decodeStream(in);
    return bitmap;
}
}
```

O código desta classe é grande, mas não se assuste, pois boa parte é referente à conexão do Google Play Services, e o restante é apenas para encapsular as APIs do Wear. Essa classe encapsula o código necessário para se conectar no Google Play Services e quando o método `onConnected(bundle)` for chamado estamos adicionando os listeners da Data API, Message API e Node API caso eles tenham sido informados pelo código cliente.

Na classe também criamos o método `putData(ath,bundle)`, que demonstra como compartilhar dados com a Data API, e o método `sendMessage(path, bytes[])`, que demonstra como enviar uma mensagem com a Message API.

O importante desse código é que para enviar uma mensagem para um dispositivo é necessário descobrir o id do nó de destino, ou seja, o id do outro dispositivo, seja smartphone ou wearable. Para isso, foi criado o método `findDeviceNodeId()`, que utiliza a Node API para buscar o id do dispositivo conectado. O id é armazenado no atributo `nodeId` da classe para depois ser utilizado para enviar as mensagens pela Message API.

Vamos prosseguir e criar um exemplo para enviar mensagens entre o smartphone e o wearable. No módulo **mobile** altere a `MainMobileActivity`, conforme demonstrado a seguir:



MainMobileActivity.java

```
public class MainMobileActivity extends BaseActivity {
    private WearUtil wearUtil;
    private int count;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main_mobile);
    }
}
```

```

        wearUtil = new WearUtil(this);
    }

    @Override
    protected void onResume() {
        super.onResume();
        wearUtil.connect();
    }

    @Override
    protected void onPause() {
        super.onPause();
        wearUtil.disconnect();
    }

    // Envia uma mensagem pela Message API
    public void onClickSendMessage(View view) {
        count++;
        wearUtil.sendMessage("/msg", new byte[] {(byte) count});
    }

    // Compartilha um Bundle com a Data API
    public void onClickPutData(View view) {
        count++;
        Bundle b = new Bundle();
        b.putString("msg", "Olá, Data API");
        b.putInt("count", count);
        // Cria o Asset
        Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.ferrari_ff);
        Asset asset = wearUtil.getAssetFromBitmap(bitmap);
        b.putParcelable("foto", asset);
        // Compartilha os dados com o wearable
        wearUtil.putData("/msg", b);
    }
}

```

A activity conecta-se ao Google Play Services com a ajuda da classe `WearUtil` e define os métodos `onClickSendMessage(view)` e `onClickPutData(view)` para enviar mensagens. Para chamar esses métodos, vamos adicionar dois botões no layout da activity:

 /mobile/res/layout/activity_main_mobile.xml

```

<LinearLayout . . .>
    <TextView android:text="@string/hello_world"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <Button
        android:layout_width="wrap_content" android:layout_height="wrap_content"

```

```

        android:text="Send Message" android:onClick="onClickSendMessage" />
    <Button
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Put Data" android:onClick="onClickPutData" />
</LinearLayout>

```

No módulo **wear** vamos fazer a outra parte do aplicativo, a qual vai receber as mensagens. Neste exemplo vamos receber uma mensagem simples pela Message API e outra que vamos ler os dados compartilhados pela Data API.

No arquivo de layout vamos adicionar um `TextView` para receber o texto enviado e um `ImageView` que vai receber o `Bitmap` (Asset) quando os dados forem enviados pela Data API. Para definir o layout da activity do wear, temos duas opções: a primeira é utilizar o layout `WatchViewStub` e especificar um layout diferente para cada tipo de tela (redonda ou quadrada). A segunda é utilizar o layout `BoxInsetLayout`, o qual permite utilizar o mesmo layout para ambos. Por padrão, o wizard gera os arquivos seguindo a primeira opção, com os layouts separados para `rect` e `round`. Mas para simplificar o exemplo estou usando a segunda opção, com um layout único.

 /wear/res/layout/activity_main_wear.xml

```

<?xml version="1.0" encoding="utf-8"?>
<android.support.wearable.view.BoxInsetLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent" android:layout_height="match_parent"
    android:padding="15dp">
    <LinearLayout android:orientation="vertical"
        android:layout_width="match_parent" android:layout_height="match_parent"
        android:padding="5dp" app:layout_box="all" >
        <TextView android:id="@+id/text"
            android:layout_width="wrap_content" android:layout_height="wrap_content" />
        <ImageView android:id="@+id/img"
            android:layout_width="wrap_content" android:layout_height="wrap_content" />
    </LinearLayout>
</android.support.wearable.view.BoxInsetLayout>

```

O próximo passo é alterar a classe `MainWearActivity` no módulo **wear**, conforme demonstrado a seguir. Observe que o código está utilizando a classe `WearUtil` da mesma forma, mas também estamos informando as interfaces `DataListener` e `MessageListener` para receber as mensagens que serão enviadas pelo smartphone.



MainWearActivity.java

```

public class MainWearActivity extends Activity implements DataApi.DataListener,
    MessageApi.MessageListener {
    private static final String TAG = "wear";
    private TextView mTextView;
    private ImageView img;
    private WearUtil wearUtil;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main_wear);
        mTextView = (TextView) findViewById(R.id.text);
        img = (ImageView) findViewById(R.id.img);
        wearUtil = new WearUtil(this);
        wearUtil.setDataListener(this);
        wearUtil.setMessageListener(this);
    }
    @Override
    protected void onResume() {
        super.onResume();
        wearUtil.connect();
    }
    @Override
    protected void onPause() {
        super.onPause();
        wearUtil.disconnect();
    }
    @Override
    public void onDataChanged(DataEventBuffer dataEventBuffer) {
        for (DataEvent event : dataEventBuffer) {
            if (event.getType() == DataEvent.TYPE_CHANGED) {
                // DataItem changed
                DataItem item = event.getDataItem();
                if (item.getUri().getPath().compareTo("/msg") == 0) {
                    // Lê a mensagem enviado pela Data API
                    DataMap dataMap = DataMapItem.fromDataItem(item).getDataMap();
                    final String msg = dataMap.getString("msg");
                    final int count = dataMap.getInt("count");
                    Asset asset = dataMap.getAsset("foto");
                    final Bitmap bitmap = wearUtil.getBitmapFromAsset(asset);
                    runOnUiThread(new Runnable() {
                        @Override

```



```

        public void run() {
            mTextView.setText(msg + "\nCount: " + count);
            img.setImageBitmap(bitmap);
        }
    });
}
} else if (event.getType() == DataEvent.TYPE_DELETED) {
    // DataItem deleted
}
}
}
@Override
public void onMessageReceived(final MessageEvent messageEvent) {
    Log.d(TAG, "onMessageReceived(): " + messageEvent.getPath());
    // Lê a mensagem
    String path = messageEvent.getPath(); // É o "/msg"
    String nodeId = messageEvent.getSourceNodeId();
    byte[] bytes = messageEvent.getData();
    final int count = bytes[0]; // Lê os bytes
    runOnUiThread(new Runnable() {
        @Override
        public void run() { // Atualiza a view
            mTextView.setText("Count: " + count);
        }
    });
}
}
}

```

Como podemos ver, a classe `WearUtil` facilita a conexão com o Google Play Services e ainda encapsula a utilização da Data API, Message API e Node API. Observe que, quando o aplicativo do wear recebe a mensagem, é utilizado o método `runOnUiThread(runnable)` para atualizar a interface, pois as mensagens são recebidas em uma thread diferente da `UiThread`.

Dica: ao lado do botão **Run** do Android Studio, você pode selecionar qual módulo será executado, neste caso o módulo `mobile` ou `wear`. Execute o módulo `mobile` em um smartphone e o módulo `wear` no emulador do wear.

A figura 39.13 mostra o aplicativo executando no smartphone. O botão **Send Message** vai enviar uma rápida mensagem, e o botão **Put Data** vai compartilhar um `Bundle` com informações.

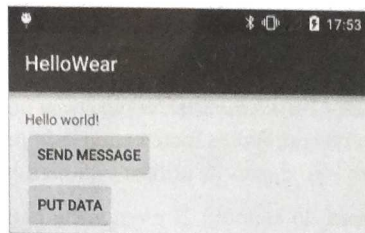


Figura 39.13 – Aplicativo mobile.

A figura 39.14 mostra o resultado no emulador do wear. A primeira parte da figura mostra a mensagem enviada pela Message API. O objetivo da Message API é trocar informações de forma rápida, como se fosse um gatilho (trigger) ou um ping que um lado da conexão pode fazer no outro. No exemplo da Message API foi passada na primeira posição do array de bytes um contador (int) que a activity no mobile fica incrementando.

A segunda parte da figura mostra os dados compartilhados pela Data API. Neste caso foi enviado o mesmo contador (int) por um `Bundle`, junto com uma string para mostrar um texto e mais uma imagem no formato `Asset`. Um `Asset` representa algum dado binário, como uma imagem. O que fizemos foi obter um `Bitmap` dos recursos do projeto e convertê-lo para o tipo `Asset`. A vantagem de utilizar assets é que a Data API faz cache deles com o objetivo de otimizar a conexão Bluetooth entre os dispositivos.

Nota: no exemplo com a Data API, note que estamos trabalhando com a classe `Bundle`, nossa velha conhecida. Mas internamente o `Bundle` é convertido para o objeto `DataMap` da API do Wear. A classe `DataMap` também funciona na estrutura de chaves e valores.

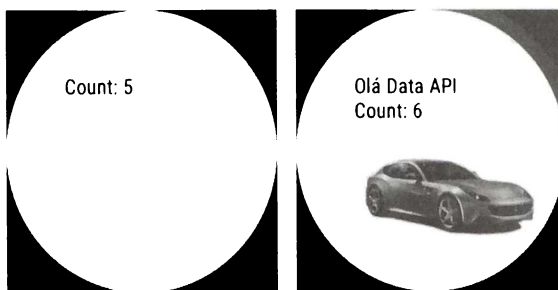


Figura 39.14 – Aplicativo wear.

39.14 Enviando uma foto tirada pela câmera para o wear

No capítulo 21, sobre multimídia, aprendemos a tirar uma foto com a câmera e fizemos o projeto *HelloCamera*. Para demonstrar como enviar imagens dinamicamente do smartphone para o relógio, vamos incrementar esse projeto e adicionar um botão para enviar a foto logo depois de utilizar a câmera.

Copie o projeto *HelloCamera* do capítulo 21 e renomeie para *HelloWearCamera*, ou se preferir abra o projeto pronto que acompanha os exemplos do livro. No layout da activity ao lado do botão de tirar foto, adicione um botão para enviar a foto. Feito isso, altere o código da activity conforme demonstrado a seguir. Note que estou mostrando apenas as partes que precisam ser alteradas.

MainActivity.java

```
import livroandroid.lib.wear.WearUtil;

public class MainActivity extends BaseActivity {
    ...
    private WearUtil wearUtil;

    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        wearUtil = new WearUtil(this);
        ...
        findViewById(R.id.btEnviarFoto).setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                // Valida se o arquivo existe
                if (file != null && file.exists()) {
                    int w = 150; // Manda a foto reduzida com 150x150 pixels
                    int h = 150;
                    Bitmap bitmap = ImageSizeUtils.getSizedImage(Uri.fromFile(file), w, h, false);
                    // Converte o bitmap para asset
                    Asset asset = wearUtil.getAssetFromBitmap(bitmap);
                    // Envia o asset para o wear
                    Bundle bundle = new Bundle();
                    bundle.putParcelable("foto", asset);
                    wearUtil.putData("/foto", bundle);
                }
            }
        });
        ...
    }
}
```

```

    . . .
    protected void onResume() {
        super.onResume();
        wearUtil.connect();
    }
    protected void onPause() {
        super.onPause();
        wearUtil.disconnect();
    }
}

```

Eu adicionei a classe `WearUtil` que estudamos anteriormente na biblioteca `android-utils`, portanto adicione esta dependência no arquivo `app/build.gradle`, assim como o Google Play Services.

```

compile 'com.google.android.gms:play-services:7.0.0'
compile 'br.com.livroandroid:android-utils:1.0.0'

```

No arquivo de manifesto declare a tag `<meta-data>` do Google Play Services.

```

<!-- Google Play Services -->
<meta-data android:name="com.google.android.gms.version"
    android:value="@integer/google_play_services_version" />

```

Agora vamos criar o módulo do **wear** no projeto. Faça isso com o wizard **File > New Module**, selecione o template do **Android Wear** e dê o nome de **wear** para o módulo. No arquivo `wear/build.gradle` declare a dependência das bibliotecas do wear e da `android-utils`.

```

compile 'com.google.android.support:wearable:1.1.0'
compile 'com.google.android.gms:play-services-wearable:7.0.0'
compile 'br.com.livroandroid:android-utils:1.0.0'

```

No arquivo de layout da activity do módulo **wear**, adicione um simples `ImageView` para receber a foto. Feito isso, altere o código da activity conforme demonstrado a seguir. Observe que estamos utilizando a classe `WearUtil` para receber os dados compartilhados pela Data API, que neste caso será a foto.



MainWearActivity.java

```

    . . .
import livroandroid.lib.wear.WearUtil;
public class MainWearActivity extends Activity implements DataApi.DataListener {
    private WearUtil wearUtil;
    private ImageView img;

```

```

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    wearUtil = new WearUtil(this);
    wearUtil.setDataListener(this);
    img = (ImageView) findViewById(R.id.img);
}
// onResume() com wearUtil.connect();
// onPause() com wearUtil.disconnect();
@Override
public void onDataChange(DataEventBuffer dataEvents) {
    for (DataEvent event : dataEvents) {
        if (event.getType() == DataEvent.TYPE_CHANGED) {
            DataItem item = event.getDataItem();
            if (item.getUri().getPath().compareTo("/foto") == 0) {
                // Recebe a foto
                DataMapItem dataMapItem = DataMapItem.fromDataItem(event.getDataItem());
                Asset photo = dataMapItem.getDataMap().getAsset("foto");
                // Converte o asset para bitmap
                final Bitmap bitmap = wearUtil.getBitmapFromAsset(photo);
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        // Mostra a foto no ImageView
                        img.setImageBitmap(bitmap);
                    }
                });
            }
        }
    }
}
}
}
}
}
}
}
}

```

Importante: observe que no lado do smartphone, antes de enviar a foto ao relógio, ela foi redimensionada para diminuir o seu tamanho. Lembrando também que o recomendado é trafegar no máximo 100kb de dados com a Data API.

A figura 39.15 mostra o resultado. Foi tirada uma foto pelo celular e enviado ao wear utilizando a Data API.



Figura 39.15 – Transferindo uma foto para o wear.

39.15 Criando views e layouts para wear

Geralmente, no wear são exibidas informações pequenas e objetivas no formato de cartões, e isso pode ser feito tranquilamente com notificações. Mas caso seja necessário é possível criar interfaces personalizadas para o wear, que podem variar de cartões, listas, grids com rolagem, alertas etc.

Para facilitar seu aprendizado, recomendo abrir o projeto **HelloWearViews** no Android Studio, o qual contém dois módulos: **mobile** e **wear**. O módulo **mobile** mostra a lista com os exemplos em um **ListView**, da mesma forma que outros projetos de exemplo do livro. A diferença é que, em vez de executar a activity no **mobile**, uma mensagem é enviada para o wearable com a Message API.

No módulo **wear** a mensagem é recebida pela Message API e disparada alguma activity ou código para demonstrar o exemplo. Acredito que você entenderá o projeto de exemplo sem problemas. No módulo **mobile**, o **ListView** é criado com o seguinte array de strings:

```
String[] items = new String[]{
    "Notification",
    "CardFragment", "CustomCardFragment", "CardFrame",
    "ListView", "ViewPager", "GridViewPager",
    "Full Screen", "Delayed Confirmation",
    "Confirmation Success", "Confirmation Error"};
```

Ao clicar em qualquer item da lista, é enviada uma mensagem para o wearable. No path (caminho) da mensagem é enviado o nome do item. Por exemplo, ao clicar no item **Notification**, é enviado o path `/Notification`. O array de bytes é sempre o mesmo, pois não será usado no exemplo.

```
public void onItemClick(AdapterView<?> parent, View view, int position, long id) {
    String item = parent.getAdapter().getItem(position).toString();
    // Envia a mensagem com o texto do item selecionado
    wearUtil.sendMessage("/") + item, new byte[] {1});
    if("ViewPager".equals(item)) {
        // Para o exemplo do ViewPager abre uma activity aqui no mobile também
        startActivity(new Intent(this,HelloViewPagerActivity.class));
    }
}
```

No módulo **wear**, a mensagem é recebida e uma activity ou algum código é iniciado para responder a mensagem. A seguir, temos um trecho de código do método `onMessageReceived(msg)` da classe `MainWearActivity`.

```
public void onMessageReceived(final MessageEvent messageEvent) {
    String path = messageEvent.getPath();
    if("/Notification".equals(path)) {
        NotificationUtil.create(this, R.mipmap.ic_launcher, "Livro Android", "Olá Wear");
    } else if("/CardFragment".equals(path)) {
        startActivity(new Intent(this, CardFragmentActivity.class));
    }
    ...
}
```

É isso! O mobile envia a mensagem para executar algum exemplo no relógio.

Nos próximos tópicos vamos estudar cada um destes exemplos. Acompanhe a explicação executando o projeto **HelloWearViews**. Você deve executar o módulo **mobile** em algum smartphone e o módulo **wear** no emulador do Android Wear.

39.16 Criando cards (cartões)

Existem várias formas de mostrar cards no wear. A primeira delas e a forma mais utilizada é por meio de notificações, que automaticamente mostram uma interface padrão com os cards.

Mas neste projeto de exemplo temos outros três exemplos, que são: `CardFragment`, `CustomCardFragment`, `CardFrame`.

O primeiro desses exemplos mostra como utilizar o fragment `CardFragment` para criar um card. Este exemplo apresenta uma activity com um layout de marcação para adicionar o fragment do cartão:

```
 /wear/res/layout/activity_card_fragment.xml

<android.support.wearable.view.BoxInsetLayout . . .>
    <FrameLayout android:id="@+id/cardLayout"
        android:layout_width="match_parent" android:layout_height="match_parent"
        app:layout_box="bottom" />
</android.support.wearable.view.BoxInsetLayout>
```

No código da activity, basta adicionar o `CardFragment` no layout como qualquer outro fragment. O método `CardFragment.create(title,text,icon)` é utilizado para criar o cartão.

```
 CardFragmentActivity.java

. . .
public class CardFragmentActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_card_fragment);
        FragmentManager fragmentManager = getFragmentManager();
        FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
        CardFragment cardFragment = CardFragment.create("Hello",
            "Meu Primeiro CardView",R.mipmap.ic_launcher);
        fragmentTransaction.add(R.id.cardLayout, cardFragment);
        fragmentTransaction.commit();
    }
}
```

Para testar esse exemplo, execute o projeto nos dois dispositivos e depois clique no item **CardFragment** na lista do smartphone. Feito isso, uma mensagem será enviada para o wear, resultando na chamada da activity `CardFragmentActivity`. O resultado pode ser visto na figura 39.16.



Figura 39.16 – CardFragment.

Caso você queira customizar a view do cartão, basta criar uma subclasse de `CardFragment` e sobrescrever o método `onCreateContentView(...)`, conforme demonstrado a seguir. Desse modo, você pode retornar o layout que quiser:

CustomCardFragment.java

```
public class CustomCardFragment extends CardFragment {
    @Override
    protected View onCreateContentView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_custom_card, container, false);
        TextView tTitle = (TextView) view.findViewById(R.id.tTitle);
        tTitle.setText(getArguments().getString("title"));
        TextView tMsg = (TextView) view.findViewById(R.id.tMsg);
        tMsg.setText(getArguments().getString("msg"));
        return view;
    }
}
```

/wear/res/layout/fragment_custom_card.xml

```
<LinearLayout android:orientation="vertical" . . . >
    <TextView android:id="@+id/tTitle" android:textColor="#000000"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
    <TextView android:id="@+id/tMsg" android:textColor="#000000"
        android:layout_width="wrap_content" android:layout_height="wrap_content" />
</LinearLayout>
```

Logo depois de criar o fragment customizado, adicione-o na activity:



CustomCardFragmentActivity.java

```
...
public class CustomCardFragmentActivity extends Activity {
    private TextView mTextView;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_card_fragment);
        FragmentManager fragmentManager = getFragmentManager();
        FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
        CustomCardFragment card = new CustomCardFragment();
        Bundle args = new Bundle();
        args.putString("title", "CardFragment");
        args.putString("msg", "Card customizado.");
        card.setArguments(args);
        fragmentTransaction.add(R.id.cardLayout, card);
        fragmentTransaction.commit();
    }
}
```

A figura 39.17 mostra o resultado. Este é o item **CustomCardFragment** do projeto de exemplo. Veja que agora você é responsável por customizar o layout, e neste caso deixei apenas um título com uma mensagem, sem nenhuma imagem, a qual existe no **CardFragment** nativo.



Figura 39.17 – CardFragment customizado.

Existe ainda outra maneira de criar um cartão customizado, que é por meio da view **CardFrame**, que pode ser inserida dentro de um **CardScrollView**.

```

wear /wear/res/layout/activity_card_frame.xml

<android.support.wearable.view.BoxInsetLayout . . >
    <android.support.wearable.view.CardScrollView
        android:id="@+id/card_scroll_view" app:layout_box="bottom"
        android:layout_height="match_parent" android:layout_width="match_parent" >
        <android.support.wearable.view.CardFrame
            android:layout_height="wrap_content" android:layout_width="match_parent">
            <LinearLayout
                android:layout_height="wrap_content" android:layout_width="match_parent"
                android:orientation="vertical" android:paddingLeft="5dp">
                <TextView android:fontFamily="sans-serif-light"
                    android:layout_height="wrap_content" android:layout_width="match_parent"
                    android:text="Título do Card" android:textColor="@color/black"
                    android:textSize="20sp"/>
                <TextView android:fontFamily="sans-serif-light"
                    android:layout_height="wrap_content" android:layout_width="match_parent"
                    android:text="Texto do Card aqui\nCom várias linhas"
                    android:textColor="@color/black" android:textSize="14sp"/>
            </LinearLayout>
        </android.support.wearable.view.CardFrame>
    </android.support.wearable.view.CardScrollView>
</android.support.wearable.view.BoxInsetLayout>

```

A figura 39.18 mostra o resultado deste exemplo. Este é o item **CardFrame** do projeto de exemplo.



Figura 39.18 – CardFrame.

Em minha opinião, sempre que possível a solução mais simples é utilizar a classe **CardFragment** que já está pronta. O código é mais simples e ele cria facilmente o visual padrão dos cartões, além de já utilizar uma imagem de ícone, dando uma boa aparência e acabamento ao cartão. Crie um cartão customizado realmente

se você precisar customizar a view, pois conforme vimos isso exige um pouco mais de código.

39.17 Criando listas

A classe `WearableListView` é utilizada para criar listas no wear e seu funcionamento é similar ao `ListView` que já conhecemos. Criar listas no wear é a parte fácil, o mais importante é entender o que você deve e o que não você deve fazer com listas, pois visualizar muitas informações no relógio naturalmente não é o ideal.

A figura 39.19 retirada da documentação oficial demonstra um princípio básico sobre listas: mostre poucas informações e com ícones grandes, ao contrário de muitas informações com ícones pequenos.

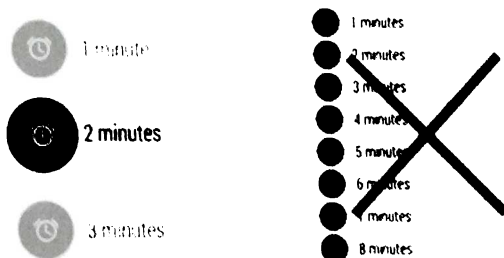


Figura 39.19 – Listas.

O exemplo que vou demonstrar vai exibir a lista de dez carros no relógio. Por questões de usabilidade, isso talvez não seja prático, mas o objetivo aqui é aprendermos a criar a lista. Para continuar, crie a classe `Carro`, conforme demonstrado a seguir:

 `Carro.java`

```
public class Carro implements Serializable {  
    public String nome;  
    public int img;  
    public Carro(String nome, int img) {  
        this.nome = nome;  
        this.img = img;  
    }  
    public static List<Carro> getListEsportivos() {  
        List<Carro> list = new ArrayList<Carro>();  
        list.add(new Carro("Ferrari FF", R.drawable.ferrari_ff));  
    }  
}
```

```

        list.add(new Carro("AUDI GT Spyder", R.drawable.audi_spyder));
        list.add(new Carro("Porsche Panamera", R.drawable.porsche_panamera));
        list.add(new Carro("Lamborghini Aventador", R.drawable.lamborghini_aventador));
        list.add(new Carro("Chevrolet Corvette Z06", R.drawable.chevrolet_corvette_z06));
        list.add(new Carro("BMW M5", R.drawable.bmw));
        list.add(new Carro("Renault Megane", R.drawable.renault_megane_trophy));
        list.add(new Carro("Maserati GranCabrio Sport", R.drawable.maserati_grancabrio_sport));
        list.add(new Carro("McLAREN MP4-12C", R.drawable.mclaren));
        list.add(new Carro("MERCEDES-BENZ C63 AMG", R.drawable.mercedes_bens));
        return list;
    }

    . . .
}

```

Esta classe contém o método `Carro.getListEsportivos()`, que retorna uma lista de carros esportivos, na qual as imagens estão fixas como recursos do projeto.

 `/wear/res/layout/activity_hello_listview.xml`

```

<android.support.wearable.view.BoxInsetLayout . . . >
    <android.support.wearable.view.WearableListView android:id="@+id/listView"
        android:layout_height="match_parent" android:layout_width="match_parent" />
</android.support.wearable.view.BoxInsetLayout>

```

No código da `activity`, utilizar o `WearableListView` é simples e basta informar o `adapter`, conceito que já estamos bem habituados.

 `HelloListViewActivity.java`

```

public class HelloListViewActivity extends Activity
    implements WearableListView.ClickListener {
    private List<Carro> carros;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_listview);
        WearableListView listView = (WearableListView) findViewById(R.id.listView);
        carros = Carro.getListEsportivos();
        listView.setAdapter(new CarroAdapter(this, carros));
        listView.setClickListener(this);
    }

    @Override
    public void onClick(WearableListView.ViewHolder v) {
        Integer position = (Integer) v.itemView.getTag();
        Carro c = carros.get(position);
    }
}

```

```

    Toast.makeText(this, "Carro: " + c.nome, Toast.LENGTH_SHORT).show();
    Intent intent = new Intent(this, CarroActivity.class);
    intent.putExtra("carro", c);
    startActivity(intent);
}
@Override
public void onTopEmptyRegionClick() {}
}

```

O código do adapter não vou mostrar no livro, pois é como qualquer outro adapter. O resultado deste exemplo pode ser visto na figura 39.20. Inclusive veja que, ao selecionar um carro, estamos navegando para outra activity, portanto os conceitos que você já conhece sobre desenvolvimento Android são aplicados no wear da mesma forma.

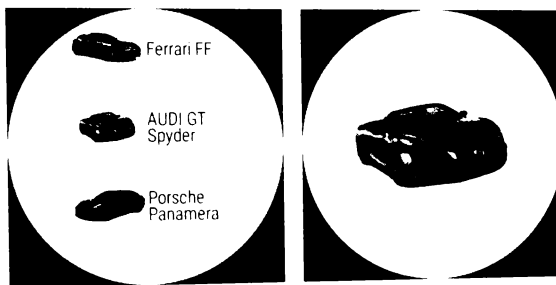


Figura 39.20 – Lista de carros com tela de detalhes.

39.18 Criando páginas (ViewPager)

O funcionamento do componente `ViewPager` é exatamente idêntico no mobile e wear. Sendo assim, não vou mostrar como criar o `ViewPager`, pois isso já sabemos. O que vamos mostrar é um exemplo clássico de galeria de fotos, a qual é visualizada e sincronizada entre o smartphone e relógio.

A figura 39.19 mostra o exemplo. A mesma lista de dez carros foi criada para preencher o `ViewPager`, portanto você pode navegar entre os carros lateralmente. O `ViewPager` foi criado tanto no aplicativo mobile quanto no wear, e, ao navegar numa página do mobile, esta página também será atualizada no wear, e vice-versa.

Isso é possível utilizando a `Message API`, pois basta monitorar o evento de troca de página do `ViewPager` para enviar uma mensagem para o outro dispositivo informando a página que ele deve mostrar.

```
viewPager.setOnPageChangeListener(new ViewPager.OnPageChangeListener() {
    @Override
    public void onPageSelected(int position) { // trocou de página
        wearUtil.sendMessage("/page", new byte[] {(byte) position});
    } ...
});
```

Do outro lado da conexão, ao receber a mensagem, o `ViewPager` é posicionado na página desejada.

```
public void onMessageReceived(MessageEvent messageEvent) {
    if("/page".equals(messageEvent.getPath())) {
        final int position = messageEvent.getData()[0];
        runOnUiThread(new Runnable() { public void run() {
            viewPager.setCurrentItem(position);
        }});
    }
}
```

Pronto, com este simples truque, temos uma galeria de fotos sincronizada entre o *mobile* e *wear* (Figura 39.21). Na figura o efeito da rolagem lateral não fica muito claro, portanto execute o projeto no emulador do wear e em algum smartphone.

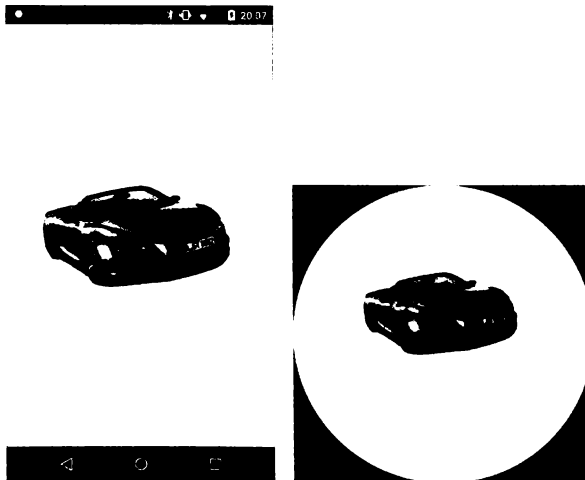


Figura 39.21 – Galeria de fotos com `ViewPager`.

39.19 Criando páginas em grid (GridViewPager)

Uma variação do `ViewPager` muito utilizada no wear é o `GridViewPager`, utilizado para criar um grid com linhas e colunas, padrão conhecido como **2D Picker**.

Esse padrão de design, também referenciado na documentação oficial por **Context Stream**, define que os cards devem ser adicionados em uma lista vertical e caso seja necessário o usuário pode fazer a rolagem para a direita para visualizar mais detalhes do cartão. A figura 39.22 demonstra esse princípio. Note que a navegação deve ir no sentido para baixo e direita.

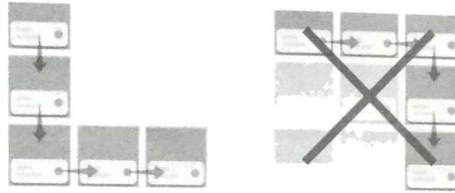


Figura 39.22 – Padrão 2D Picker.

Não é exatamente isso que faremos no próximo exemplo, mas nosso objetivo é apenas didático para mostrar como criar esse grid com cards. No exemplo que vamos criar, teremos um grid de dez linhas com três colunas com os carros (clássicos, esportivos e luxos). Cada linha terá um card com informações do carro.

Então, vamos lá! No layout da activity, basta adicionar o `GridViewPager`.

 /wear/res/layout/activity_hello_gridviewpager.xml

```
<android.support.wearable.view.BoxInsetLayout . . . >
    <android.support.wearable.view.GridViewPager android:id="@+id/viewPager"
        android:layout_width="match_parent" android:layout_height="match_parent"
        android:keepScreenOn="true" />
</android.support.wearable.view.BoxInsetLayout>
```

Dica: neste layout utilizei o atributo `keepScreenOn="true"`, que faz a tela do relógio não apagar enquanto a view estiver visível. Isso pode ser útil em determinados casos.

No código vamos utilizar um adapter customizado para preencher o `GridViewPager`. O adapter vai receber as três listas de carros.

 HelloGridViewPagerActivity.java

```
public class HelloGridViewPagerActivity extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_gridviewpager);
        List<Carro> classicos = Carro.getListClassicos(); // dez carros clássicos
        List<Carro> esportivos = Carro.getListEsportivos(); // dez carros esportivos
        List<Carro> luxo = Carro.getListLuxo(); // dez carros luxo
        GridViewPager grid = (GridViewPager) findViewById(R.id.viewPager);
        grid.setAdapter(new CarrosGridPagerAdapter(this, getFragmentManager(), classicos,
            esportivos, luxo));
    }
}
```

O adapter deve herdar de `FragmentGridPagerAdapter` e implementar os métodos `getRowCount()`, `getColumnCount(row)` e `getFragment(row,col)`.

 CarrosGridPagerAdapter.java

```
public class CarrosGridPagerAdapter extends FragmentGridPagerAdapter {
    private final LayoutInflater mInflater;
    private Context context;
    private List<Carro> classicos;
    private List<Carro> esportivos;
    private List<Carro> luxo;
    public CarrosGridPagerAdapter(Context context,FragmentManager fm, List<Carro> classicos,
        List<Carro> esportivos,List<Carro> luxo) {
        super(fm);
        // inicializa os atributos...
    }
    public Fragment getFragment(int row, int col) {
        Carro c = classicos.get(row);
        if(col == 1) { c = esportivos.get(row); }
        else if(col == 2) { c = luxo.get(row); }
        CarroCardFragment card = new CarroCardFragment();
        Bundle args = new Bundle();
        args.putString("nome",c.nome);
        args.putInt("img",c.img);
        card.setArguments(args);
        return card;
    }
}
```

```

    public int getRowCount() { return classicos.size(); }
    public int getColumnCount(int row) { return 3; }
}

```

O método `getColumnCount(row)` deve retornar a quantidade de colunas para cada linha. Neste caso deixei fixas as três colunas (clássicos, esportivos e luxo). O método `getRowCount()` deve retornar a quantidade de linhas do grid, que neste caso são dez carros. O método `getFragment(row,col)` é o mais importante e deve retornar um fragment que será inserido em cada posição do grid. Poderíamos ter usado o `CardFragment` padrão com esse código, caso fosse necessário.

```
CardFragment card = CardFragment.create("Carro", c.nome, c.img);
```

Porém, as imagens dos carros que coloquei no projeto são muito grandes, então criei um fragment específico para o cartão dos carros. No layout desse cartão deixei apenas o nome e a foto do carro. O `ImageView` que vai mostrar a foto está configurado com o atributo `scaleType="fitCenter"` para redimensionar a foto e centralizá-la no espaço disponível do cartão.

 /wear/res/layout/fragment_carro_card.xml

```

<LinearLayout android:orientation="vertical" . . . >
    <TextView android:id="@+id/tNome" android:gravity="center"
        android:layout_width="match_parent" android:layout_height="wrap_content" />
    <ImageView android:id="@+id/img" android:scaleType="fitCenter"
        android:layout_width="match_parent" android:layout_height="match_parent" />
</LinearLayout>

```

Na classe `CarroCardFragment` é criado o cartão customizado com esse layout.

 CarroCardFragment.java

```

public class CarroCardFragment extends CardFragment {
    @Override
    protected View onCreateContentView(LayoutInflater inflater, ViewGroup container,
        Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.fragment_carro_card, container, false);
        TextView tNome = (TextView) view.findViewById(R.id.tNome);
        tNome.setText( getArguments().getString("nome"));
        ImageView img = (ImageView) view.findViewById(R.id.img);
        img.setImageResource( getArguments().getInt("img"));
        return view;
    }
}

```

O resultado deste exemplo pode ser visto na figura 39.23, que mostra o grid com as três colunas. Lembrando que isso é um grid com dez linhas e três colunas, portanto faça a rolagem para entender o funcionamento.

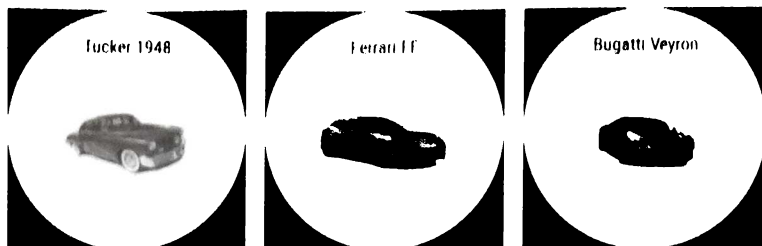


Figura 39.23 – GridViewPager com carros.

Apenas tenha atenção, pois embora esse exemplo tenha demonstrado como criar grids no padrão **2D Picker**, não estamos seguindo exatamente um princípio importante de design de aplicativos para o wear, chamado de **ContextStream** (procure por este termo no site do Android Design). Segundo esse princípio, os cards devem ser adicionados em uma lista vertical e caso seja necessário o usuário pode fazer a rolagem para a direita para visualizar mais detalhes do cartão. A figura 39.24 demonstra este princípio. Mas, agora você já deve conseguir fazer isso. Seria um bom exercício, o que acha?

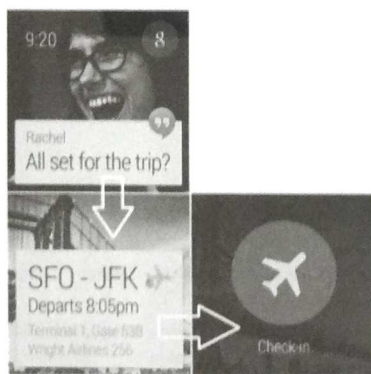


Figura 39.24 – Cards em grid.

Nota: cards utilizam imagens de fundo para melhorar o visual, trazendo um conteúdo responsivo e significativo para cada cenário. Para configurar uma imagem de fundo, utilize o atributo `android:background` de qualquer layout.

39.20 Aplicativos em tela cheia (Full-Screen)

Por padrão os aplicativos para wear utilizam os temas `Theme.DeviceDefault` ou `Theme.DeviceDefault.Light`, e para remover um cartão ou fechar um aplicativo é preciso fazer o gesto de swipe da esquerda para a direita, conhecido como *Swipe to dismiss*.

Se por algum motivo você precisa desabilitar esse gesto padrão para deixar uma activity em tela cheia, basta criar um tema customizado, atribuindo a propriedade `windowSwipeToDismiss` para `false`.

```
<style name="FullScreenTheme" parent="android:Theme.DeviceDefault.Light">
    <item name="android:windowSwipeToDismiss">false</item>
</style>
```

No projeto de exemplo do livro, criei a classe `FullScreenActivity` e configurei esta activity para utilizar este tema customizado.

```
<activity android:name=".FullScreenActivity" android:theme="@style/FullScreenTheme" />
```

Como essa activity desabilitou o gesto padrão para fechar o aplicativo, para sair do aplicativo é preciso utilizar a view `DismissOverlayView`, que permite ao usuário tocar na tela e segurar por dois segundos para fechá-la. Ao fazer isso, será exibido um alerta de confirmação para o usuário. Segundo as boas práticas de interface para o wear, caso você utilize este tipo de recurso para fechar a tela, é recomendado informar ao usuário no início do aplicativo que para fechar a tela é necessário tocar e segurar por dois segundos.

Então vamos ao código. No layout da activity, basta adicionar um `DismissOverlayView` por cima de tudo. Lembrando que o layout `BoxInsetLayout` é uma subclasse de `FrameLayout`, portanto ele vai empilhar as views.

 /wear/res/layout/activity_full_screen.xml

```
<android.support.wearable.view.BoxInsetLayout . . >
    <TextView
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_gravity="center" android:text="Tela em Full Screen" />
    <android.support.wearable.view.DismissOverlayView
        android:id="@+id/dismiss_overlay"
        android:layout_width="match_parent" android:layout_height="match_parent" />
</android.support.wearable.view.BoxInsetLayout>
```

Para utilizar o `DismissOverlayView` na activity, é preciso criar um detector de gestos e sobrescrever o método `onTouchEvent(ev)`, conforme visualizado a seguir.

 CarroCardFragment.java

```
public class FullScreenActivity extends Activity {
    private GestureDetector mDetector;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_full_screen);
        final DismissOverlayView d = (DismissOverlayView) findViewById(R.id.dismiss_overlay);
        d.setIntroText("Para sair, clique e segure.");
        // Mostra o texto de introdução na primeira vez
        d.showIntroIfNecessary();
        // Configura o detector de gestos
        mDetector = new GestureDetector(this, new GestureDetector.SimpleOnGestureListener() {
            public void onLongPress(MotionEvent ev) {
                d.show(); // Mostra o alerta para sair
            }
        });
    }
    @Override
    public boolean onTouchEvent(MotionEvent ev) {
        return mDetector.onTouchEvent(ev) || super.onTouchEvent(ev);
    }
}
```

O resultado deste exemplo pode ser visto na figura 39.25. Na primeira parte da figura, podemos ver o alerta inicial que é exibido para o usuário ao abrir o aplicativo pela primeira vez. Logo depois, podemos ver que a aplicação mostrou o layout da activity normalmente. A terceira parte da figura mostra o alerta que é exibido quando o usuário toca na tela e segura por dois segundos, permitindo assim que ele feche o aplicativo. Esse padrão é conhecido como **Long press to dismiss**.

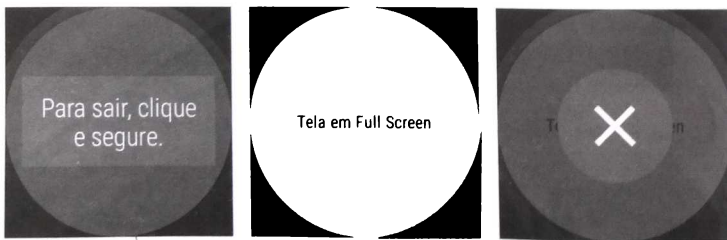


Figura 39.25 – Toque longo para sair.

Nota: dependendo do modo de interação e gestos utilizados pelo seu aplicativo, como para utilizar um grid ou um mapa, o gesto de `swipe to dismiss` utilizado por padrão para fechar o aplicativo pode interferir no funcionamento da tela. Justamente por isso, alguns aplicativos optam por desabilitar esse gesto padrão, em troca de sair por meio do padrão `Long press to dismiss`.

39.21 Animação de confirmação

No *wear*, um padrão importante são os alertas de confirmação com um tempo de espera (`delay`). Durante determinado intervalo de tempo, o aplicativo mostra um alerta com uma animação, sendo que o usuário pode intervir cancelando a execução, ou deixar que o aplicativo execute a ação depois do tempo estipulado.

A figura 39.26 mostra esse alerta com a animação. Ao redor da imagem um círculo fica sendo animado até preencher toda a área (execute no emulador para conferir a animação). Na parte direita da imagem, podemos ver uma notificação que criei no exemplo para informar que o tempo da animação terminou e a ação pode ser executada.

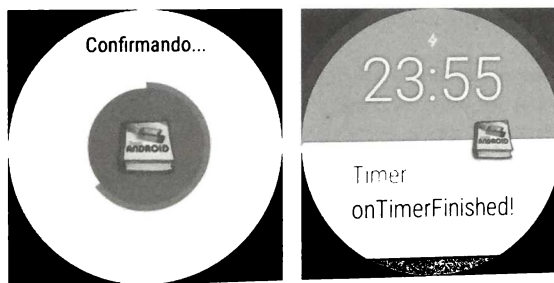


Figura 39.26 – Animação de confirmação.

Para criar esse tipo de animação antes de executar uma ação, basta adicionar uma view do tipo `DelayedConfirmationView` no layout.

```
res/layout/activity_confirmation_delayed.xml

<android.support.wearable.view.BoxInsetLayout . . . >
    <TextView android:layout_gravity="center_horizontal"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="Confirmando..." />
```

```

<android.support.wearable.view.DelayedConfirmationView
    android:id="@+id/delayed_confirmation"
    android:layout_width="wrap_content" android:layout_height="wrap_content"
    android:layout_gravity="center" android:src="@drawable/ic_launcher"
    app:circle_color="@color/blue" app:circle_border_color="@color/red"
    app:circle_radius="@dimen/circle_radius"
    app:circle_radius_pressed="@dimen/circle_radius_pressed"
    app:circle_padding="@dimen/circle_padding"
    app:circle_border_width="@dimen/circle_border_normal_width" />
</android.support.wearable.view.BoxInsetLayout>

```

No código o método `setTotalTimeMs(long)` recebe o tempo em milissegundos que a animação deve durar e o método `start()` inicia a animação. Para receber o resultado da animação (se o usuário interrompeu ou não a execução), deve-se implementar a interface `DelayedConfirmationListener`. O método `onTimerSelected(view)` é chamado se o usuário tocar na view da animação, indicando que a ação deve ser interrompida. Já o método `onTimerFinished(view)` é chamado se o usuário não fizer nada, indicando que a ação deve executar normalmente.



ConfirmationDelayedActivity.java

```

public class ConfirmationDelayedActivity extends Activity
    implements DelayedConfirmationView.DelayedConfirmationListener {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_confirmation_delayed);
        DelayedConfirmationView d = (DelayedConfirmationView) findViewById(R.id.delayed_confirmation);
        d.setTotalTimeMs(5 * 1000); // cinco segundos
        d.start();
        d.setListener(this);
    }
    @Override
    public void onTimerFinished(View view) { // usuário não fez nada
        NotificationUtil.create(this, R.mipmap.ic_launcher, "Timer", "onTimerFinished!");
        finish();
    }
    @Override
    public void onTimerSelected(View view) { // usuário tocou na animação
        NotificationUtil.create(this, R.mipmap.ic_launcher, "Timer", "onTimerSelected!");
        finish();
    }
}

```

39.22 Alertas de sucesso e erro

No wear, também é muito comum ver alertas com mensagens de sucesso e erro quando o usuário executa determinadas ações. Para criar esse tipo de alerta, deve-se adicionar a activity nativa `ConfirmationActivity` no arquivo de manifesto:

```
<application>
...
<activity
    android:name="android.support.wearable.activity.ConfirmationActivity" />
</application>
```

No código para criar as mensagens, basta disparar uma intent com essa activity. O parâmetro `EXTRA_MESSAGE` recebe o texto da mensagem e o parâmetro `EXTRA_ANIMATION_TYPE` corresponde ao tipo do alerta, que pode ser mensagem de sucesso `SUCCESS_ANIMATION`, mensagem de erro `FAILURE_ANIMATION` ou uma mensagem para indicar que alguma activity foi aberta no smartphone `OPEN_ON_PHONE_ANIMATION`.

Na classe `MainWearActivity` do projeto **HelloWearViews**, demonstrei como criar as animações de sucesso e erro, conforme mostra a figura 39.27. A seguir podemos ver o código que cria essas mensagens.

```
public void onMessageReceived(final MessageEvent messageEvent) {
    String msg = messageEvent.getPath();
    ...
    else if("/Confirmation Success".equals(msg)) {
        Intent intent = new Intent(this, ConfirmationActivity.class);
        intent.putExtra(ConfirmationActivity.EXTRA_ANIMATION_TYPE,
            ConfirmationActivity.SUCCESS_ANIMATION);
        intent.putExtra(ConfirmationActivity.EXTRA_MESSAGE, "Mensagem de sucesso!");
        startActivity(intent);
    } else if("/Confirmation Error".equals(msg)) {
        Intent intent = new Intent(this, ConfirmationActivity.class);
        intent.putExtra(ConfirmationActivity.EXTRA_ANIMATION_TYPE,
            ConfirmationActivity.FAILURE_ANIMATION);
        intent.putExtra(ConfirmationActivity.EXTRA_MESSAGE, "Mensagem de erro!");
        startActivity(intent);
    }
}
```

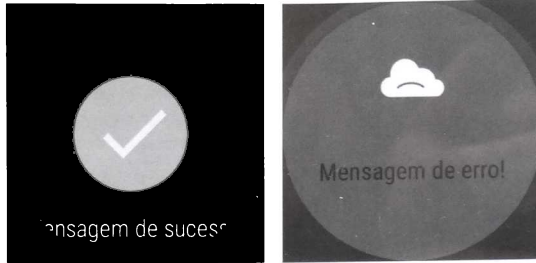



Figura 39.27 – Alerta de sucesso e erro.

39.23 Interceptando eventos em background

Se você reparou, os exemplos que fizemos com a Node API, Message API e Data API, todos utilizaram a classe `WearUtil` para se conectar ao Google Play Services e ativar os listeners de cada API. Isso foi feito dentro de uma `activity`; assim é obrigatório que o aplicativo do relógio esteja aberto para receber as mensagens.

Entretanto, isso muitas vezes não é o desejado no caso de aplicativos para relógios, tendo em vista que na maioria das vezes eles ficam fechados. Sendo assim, temos de ter uma maneira de receber essas mensagens em background, pois o relógio deve acordar e mostrar uma notificação relevante ao usuário em momentos chaves do dia. Esse é um princípio de interface e design importante do wear.

Felizmente, isso é simples de fazer. Na API do Wear, existe um service especial chamado `WearableListenerService`, que já implementa as interfaces dos listeners da Node API, Message API e Data API. A classe `WearableListenerService` é filha direta da classe `Service`, a qual já estudamos, portanto ela fica executando em segundo plano.

O segredo é criar uma subclasse dessa classe e cadastrá-la no arquivo de manifesto da seguinte forma:

AndroidManifest.xml

```
<service android:name=".HelloListenerService" >
    <intent-filter>
        <action android:name="com.google.android.gms.wearable.BIND_LISTENER" />
    </intent-filter>
</service>
```

A seguir, podemos ver o código desta subclasse de `WearableListenerService`:

```
 HelloListenerService.java

public class HelloListenerService extends WearableListenerService {
    @Override
    public void onMessageReceived(MessageEvent messageEvent) {
        super.onMessageReceived(messageEvent); // Message API
    }
    @Override
    public void onDataChanged(DataEventBuffer dataEvents) {
        super.onDataChanged(dataEvents); // Data API
    }
    @Override
    public void onPeerConnected(Node peer) {
        super.onPeerConnected(peer); // Node API
    }
    @Override
    public void onPeerDisconnected(Node peer) {
        super.onPeerDisconnected(peer); // Node API
    }
}
```

Pronto! O simples fato de criar esse service e registrá-lo no arquivo de manifesto já é suficiente para receber mensagens do aplicativo que está no smartphone. As mensagens serão recebidas mesmo com o aplicativo do relógio fechado. Se quiser testar a teoria, veja o exemplo `WearableListenerService`.

39.24 Localização e sensores

Os dispositivos com wear têm sensores como acelerômetro, contador de passos, batimentos cardíacos e GPS. Porém, não é garantido que cada dispositivo tenha todos os sensores, por isso é importante checar para ver se determinado sensor existe antes de utilizá-lo.

Por exemplo, o primeiro Moto 360 não tem o sensor de GPS, enquanto o Samsung Gear tem. O seguinte código pode ser utilizado para detectar se o relógio tem o sensor do GPS:

```
boolean hasGps = getPackageManager().hasSystemFeature(PackageManager.FEATURE_LOCATION_GPS);
```

Caso o retorno seja verdadeiro, podemos utilizar a Location API do Google Play Services, conforme já estudamos, pois o funcionamento é idêntico ao do smartphone.

Os restantes dos sensores também funcionam da mesma forma, sendo assim podemos utilizar a classe `SensorManager`, conforme também já estudamos.

Sobre este assunto, vale ressaltar que segundo as boas práticas do Google recomenda-se utilizar o sensor do smartphone sempre que possível, caso os dispositivos estejam pareados. Veja mais detalhes a respeito disso na apresentação **Wear & Sensors (Android Performance Patterns)** disponível no canal do YouTube do Google Developers. Segundo o Google, a justificativa para preferir a utilização dos sensores do smartphone é para economizar a bateria do relógio, até porque o smartphone tem uma bateria melhor e um processamento muito maior. Enviar mensagens com os dispositivos pareados pode custar menos do que ativar os sensores no relógio. Por exemplo, se você precisa do sensor de contador de passos é possível utilizar o sensor do smartphone e enviar o valor para o relógio por meio da Message API. Porém, caso os dispositivos não estejam pareados, justifica-se utilizar diretamente o sensor do relógio. É o caso de um aplicativo de corrida, no qual o relógio pode ser utilizado para acompanhar a quantidade de passos, batimentos cardíacos e coordenadas GPS sem a necessidade de estar pareado com o smartphone.

39.25 Links úteis

Neste capítulo estudamos o desenvolvimento para o Android Wear. O assunto é novo, empolgante e ainda pouco explorado. Como tudo que é novidade, o mundo dos wearables ainda vai crescer muito, portanto esteja preparado. Seguem alguns links para continuar seus estudos.

- **Android.com – Android Wear**

<http://www.android.com/wear/>

- **Android Developers – Android Wear**

<https://developer.android.com/wear>

- **Android Design – Android Wear**

<https://developer.android.com/design/wear>



CAPÍTULO 40

Google Play

Neste capítulo, vamos verificar os passos necessários para publicar uma aplicação no Google Play.

Se você chegou até aqui, parabéns. Espero que tenha gostado do livro. Desde o início da leitura, fomos do básico ao avançado e vimos muitos exemplos práticos, mas isso é só o começo, e espero que você não pare por aqui. O próximo passo é continuar seus estudos, principalmente lendo a documentação oficial e os guidelines de interface, para criar aplicativos que encantem os usuários.

40.1 Controle da versão de sua aplicação

Antes de publicar um aplicativo no Google Play, é necessário definir a versão do aplicativo no arquivo *app/build.gradle*, o que é feito configurando os itens *versionCode* e *versionName*:



app/build.gradle

```
android {  
    . . .  
    defaultConfig {  
        applicationId "br.com.livroandroid.carros"  
        minSdkVersion 9 // API Level mínima  
        targetSdkVersion 22 // API Level target/alvo (sempre a última)  
        versionCode 1 // Número identificador da versão no Google Play  
        versionName "1.0" // Versão que o usuário vai ver.  
    }  
    . . .  
}
```

O item *versionCode* é utilizado pelo Google Play para instalar ou atualizar um aplicativo pela loja, portanto esse número sempre deve ser maior que o da versão

anterior. O item `versionName` é mostrado ao usuário para que ele saiba a versão em que a aplicação está.

Se necessário, podemos recuperar a versão dentro do código:

```
try {
    PackageInfo packageInfo = getPackageManager().getPackageInfo(
        "br.com.livroandroid.carros", 0);
    Log.d(TAG, "Pacote: " + packageInfo.packageName + ", versão: " +
        packageInfo.versionCode + " - " + packageInfo.versionName);
} catch (NameNotFoundException e) {
    Log.e(TAG, e.getMessage(), e);
}
```

Esse código imprimirá a seguinte mensagem no LogCat:

```
D/livro(328): Pacote: br.com.livroandroid.carros, versão: 1 - 1.0
```

40.2 Compilando o projeto corretamente

Outra configuração importante no arquivo *app/build.gradle* diz respeito à compatibilidade da aplicação com a versão da imagem do sistema instalada no dispositivo.

Sempre configure o atributo `minSdkVersion` para a menor versão que o aplicativo vai suportar e o atributo `targetSdkVersion` para a última versão, para otimizar o build e ativar os recursos dos novos dispositivos. O Google Play vai respeitar a configuração do `minSdkVersion` no momento da instalação; caso o dispositivo não seja compatível (API Level mínima), a aplicação não será instalada.

Nota: a biblioteca de compatibilidade `Support Library`, criada para trazer a compatibilidade com as versões antigas do Android, apresenta vários recursos, como `action bar`, `Toolbar`, `fragments`, `ViewPager`, `notifications`, animações, entre muitos outros. A biblioteca de compatibilidade `appcompat-v7` é compatível com API Level 7 (Android 2.1). Atualmente, eu costumo criar aplicativos com compatibilidade com API Level 9 (Android 2.3). Mas isso depende do aplicativo que você está desenvolvendo. Se seu cliente for algum grande banco, é claro que eles vão querer atingir o máximo de usuários possíveis; porém, muitas vezes, dependendo do aplicativo, pode-se definir a compatibilidade com Android 5.x ou superior, tudo depende dos requisitos do projeto.

Para informações sobre o número de cada API Level, visite esta página:

<http://developer.android.com/guide/appendix/api-levels.html>

Para informações sobre a biblioteca de compatibilidade, visite esta página:

<http://developer.android.com/tools/support-library/>

40.3 Assinando o aplicativo pelo Android Studio/Gradle

No capítulo 38, sobre Gradle, aprendemos a criar o certificado keystore de **release** e assinamos o projeto com esse certificado. Para assinar o aplicativo, podemos utilizar o menu **Build > Generate Signed APK** ou fazer o build pelo próprio Gradle, pois temos os dois tipos de builds: **debug** e **release**.

Enfim, isso é só um lembrete, pois já aprendemos a fazer isso no capítulo 38. Apenas lembre-se de que para publicar no Google Play deve ser utilizado o certificado de **release**.

Dica: nunca perca o certificado de **release**, pois somente com ele é possível atualizar um aplicativo na loja do Google Play.

Embora seja mais fácil assinar o aplicativo com o Android Studio e o Gradle, também é possível assinar o aplicativo utilizando as ferramentas **keytool** e **jarsigner** disponíveis no JDK. Caso precise de mais detalhes sobre isso, recomendo ler a documentação oficial.

<http://developer.android.com/tools/publishing/app-signing.html>

40.4 Publicando no Google Play

A publicação de uma aplicação no Google Play é simples: basta entrar no seguinte site, utilizando um login válido do Gmail:

<https://play.google.com/apps/publish/>

Será necessário pagar US\$ 50,00 para abrir sua conta. Depois disso, você poderá fazer o upload de diversas aplicações. O pagamento é único e não precisa renovar.

Depois que sua conta for criada, basta entrar no site de administração do Google Play para visualizar as aplicações já publicadas ou publicar uma nova.

Nota: o cadastro de desenvolvedor para publicar um aplicativo no Google Play custa US\$ 50,00, sendo necessário um cartão de crédito internacional para fazer o pagamento.

Na página de console do Google Play, você pode clicar no botão **Add New Application** para preencher o formulário com uma breve descrição de sua aplicação e enviar alguns screenshots. No formulário, basta fazer o upload do arquivo *.apk* que foi assinado, escolher a categoria da aplicação (Jogo, Finanças, Saúde, Esportes, Educação etc.) e definir se ela é gratuita ou paga. O formulário que você precisa preencher é simples, e há uma boa documentação de cada campo a ser preenchido, então você não terá dificuldades.

Depois de publicar uma aplicação, ela será disponibilizada quase que instantaneamente no Google Play e estará pronta para ser instalada.

Lembre-se de que, a cada atualização da aplicação, é necessário incrementar o parâmetro *versionCode* no arquivo *build.gradle*. Mas a boa notícia é que se você tentar publicar um *apk* sem incrementar o número da versão, o Google Play vai avisá-lo, então é tudo tranquilo.

Depois de publicar, lembre-se de acompanhar os comentários e relatório de erros enviados pelos usuários, para corrigir eventuais bugs e a fim de melhorar seu aplicativo. Ao atualizar o aplicativo na loja, nunca se esqueça de fornecer uma breve descrição com as melhorias da nova versão, para o usuário ter conhecimento das novidades.

Nota: para aplicativos pagos, o desenvolvedor recebe 70% do valor do aplicativo em cada venda, e o Google fica com 30% para manter o serviço.

40.5 Monetização com anúncios

Ao publicar aplicativos no Google Play, talvez algo que você tenha interesse em aprender é como colocar os famosos anúncios no aplicativo, a fim de tirar um lucro com o aplicativo, mesmo se ele for gratuito.

Eu mesmo nunca coloquei nenhum anúncio, pois na empresa fazemos aplicativos comerciais em forma de projeto fechado para nossos clientes, então não posso lhe dizer se realmente os anúncios dão dinheiro ou não. O que todos sabem é que, caso seu aplicativo faça sucesso e seja baixado por milhões de pessoas no mundo inteiro, então com certeza você ficará bem feliz. O que você precisa é ter uma boa ideia.

Caso tenha interesse em aprender mais sobre anúncios, recomendo assistir duas palestras feitas no Google I/O. No Google I/O 2010 foi apresentada a plataforma **AdMob** e todos os conceitos foram explicados durante uma palestra de 40 minutos. Esta palestra pode ser encontrada no YouTube ou no site do Google I/O, basta procurar pelo

título “Google I/O 2010 – Analyzing and monetizing your mobile apps”. É extremamente recomendado assistir o vídeo, pois dados muito interessantes, gráficos e números são exibidos. Como se não bastasse, durante o Google I/O 2011 novamente uma grande palestra foi feita sobre o tema, com o título “Don’t just build a mobile app. Build a business”. E novamente recomendo que você assista o vídeo.

Durante a palestra foram explicadas as diversas formas de rentabilizar a sua aplicação, conforme a figura 40.1 que foi extraída de um dos slides da apresentação.

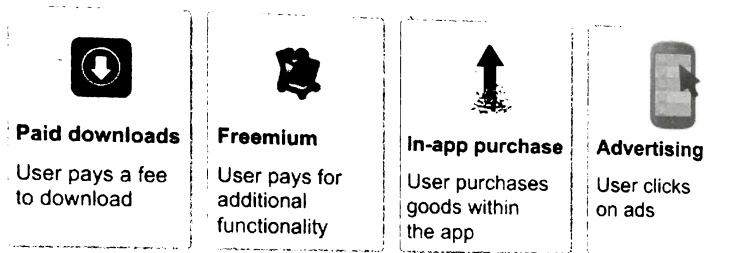


Figura 40.1 – Apresentação Google I/O 2011.

Basicamente, existem quatro maneiras para rentabilizar uma aplicação:

Parâmetro	Descrição
Paid downloads	Este é o tradicional método em que o usuário paga para efetuar o download da aplicação.
Premium	Aqui os usuários podem instalar a aplicação gratuitamente, mas pagam para adquirir mais funcionalidades.
In-app billing	Esta é outra maneira de disponibilizar a aplicação de forma gratuita, mas cobrar por compras dentro do aplicativo utilizando a plataforma integrada do Google Play. No caso de um jogo, poderiam ser comprados novas fases e poderes especiais. E no caso de uma aplicação poderiam ser adquiridos novos módulos e recursos.
Advertising	Neste método a aplicação é disponibilizada de forma gratuita, mas anúncios de publicidade são adicionados na aplicação.

Sobre criar anúncios, chamados de "Ads", é utilizado o **Google Mobile Ads SDK**. Na documentação oficial tem um passo a passo (quick start) que mostra como criar os anúncios. É simples e rápido, caso você precise.

<https://apps.admob.com/admob/>

<http://developer.android.com/google/play-services/ads.html>

<https://developers.google.com/mobile-ads-sdk/docs/admob/android/quick-start>

Sobre o **In-app billing**, também recomendo estudar a documentação oficial:

<http://developer.android.com/google/play/billing/>

Lembre-se também de que mostramos como criar builds customizados com o Gradle, portanto você pode criar versões gratuitas e pagas do seu aplicativo.

40.6 Links úteis

Chegamos ao final do livro e gostaria de parabenizá-lo pela grande conquista de se tornar um desenvolvedor Android. Agora, você está pronto para entrar em um dos mercados que mais crescem no mundo.

Desejo-lhe boa sorte nessa jornada e aguardo os seus aplicativos no Google Play!

Para continuar seus estudos sobre como assinar o aplicativo e publicá-lo na loja, segue o link com a documentação oficial.

- **Android Developers – Signing Your Applications**

<http://developer.android.com/tools/publishing/app-signing.html>

Conheça também do mesmo autor



O objetivo deste livro é ensinar o desenvolvimento de aplicativos para iPhone e iPad. Explica os conceitos desde o básico, com uma introdução ao Xcode e a linguagem Objective-C, desenvolvimento para iOS e componentes visuais que podem ser utilizados. Aborda diversos temas e boas práticas de desenvolvimento, comunicação com web services, XML e JSON, gerenciamento de threads, banco de dados e arquivos, mapas e GPS, recursos de multimídia, animações, sensores de acelerômetro e giroscópio, gerenciamento de memória no Objective-C e ferramentas de profiler disponíveis no Xcode.

Google ANDROID

O Android é atualmente o sistema operacional móvel mais utilizado no mundo e está disponível para diversas plataformas, como smartphones, tablets, TV (Google TV), relógios (Android Wear), óculos (Google Glass) e carros (Android Auto).

Este livro convida o leitor a mergulhar no incrível mundo do Android, onde a imaginação é o limite. A obra traz todos os passos necessários para desenvolver aplicativos para smartphones, tablets e relógios, desde o básico – sobre a instalação do emulador e configuração do ambiente de desenvolvimento no Android Studio – até aplicações que utilizem recursos avançados, como mapas, localização por GPS, multimídia, consulta em web services, persistência em banco de dados, sensores, execução de serviços em segundo plano e muito mais.

Será desenvolvido um aplicativo completo passo a passo, conforme as boas práticas de desenvolvimento e interface e utilizando conceitos do Material Design.

Para mais informações, visite o site do livro
www.livroandroid.com.br.

Ricardo R. Lecheta é formado em Ciências da Computação e pós-graduado em Gestão do Desenvolvimento de Software pela PUC-PR. Atualmente, trabalha com desenvolvimento e consultoria de tecnologias mobile para diversas plataformas.

Pode ser contatado pelo email rlecheta@gmail.com.

Fique conectado:



twitter.com/novateceditora



facebook.com/novatec



www.novatec.com.br

ISBN 978-85-7522-440-3



9 788575 224403